MASTER IN COMPUTER SCIENCE

Définition d'une architecture générique de déploiement d'applications dans un cadre distribué

Canart, Sébastien

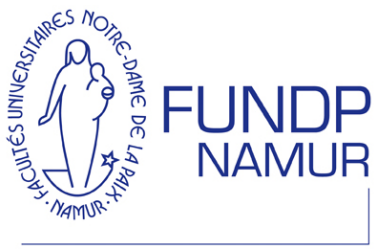
Award date:
2009

Awarding institution:
University of Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Faculté d'Informatique
Année académique 2008-2009

Définition d'une architecture générique de déploiement d'applications dans un cadre distribué

Sébastien CANART



Mémoire présenté en vue de l'obtention du grade de maître en informatique.

Résumé

Ce mémoire est une proposition pour une architecture permettant le déploiement et l'utilisation d¿applications dans un cadre distribué. Cette architecture a été conçue pour traiter spécifiquement des applications complexes composées d'un grand nombre d'unités d'exécution interconnectées. Le déploiement consiste alors en l'installation et l'interconnexion des dites unités sur une infrastructure constituée de *machines* connectées physiquement. Notre proposition consiste en une architecture qui automatise ce processus. Nous discuterons ensuite des différents choix que nous avons réalisés lors de la conception de cette architecture. Nous allons ensuite présenter le prototype qui a été réalisé et qui l¿implémente. Nous terminerons par quelques commentaires et pistes de développement futur.

Abstract

This thesis is a proposal for an architecture enabling the deployment and utilization of applications in a distributed framework. This architecture has been designed to specifically handle complex applications made of a large number of interconnected execution units. The deployment comprises the installation and interconnection of these units on an infrastructure made of hosts physically connected. Our proposal is about a new architecture which automates this process. We will discuss about several choices that we made when creating this architecture. We will then present the prototype implemented. We will conclude with a few comments and ideas for future development.

Remerciements

Nous souhaitons remercier les personnes qui nous ont supporté le long de la rédaction de ce mémoire.

Nous souhaitons remercier particulièrement Monsieur Vincent Englebert (Université de Namur) pour son aide et ses remarques.

Nous souhaitons également remercier Monsieur Antoine Beugnard, Monsieur Fabien Dagnat, Monsieur Jérémy Buisson et Monsieur Trinh Anh Tuan pour leur aide, leurs conseils lors de notre stage à Brest.

De plus, nous souhaitons remercier Madame Armelle Lannuzelle ainsi que le département informatique de Télécom Bretagne pour leur accueil chaleureux lors de notre stage à Brest.

Pour terminer, nous souhaitons remercier la famille pour le support lors de la rédaction de ce mémoire.

Table des matières

Table des figures

Chapitre 1

Introduction

L'informatique a beaucoup évolué ces dernières années. Cette évolution, tant au niveau matérielle que logicielle, permet aux ordinateurs de faire tourner des applications de plus en plus complexes. La capacité de communication entre les machines permet maintenant d'utiliser un grand nombre d'ordinateurs afin d'utiliser leur puissance collective pour effectuer des calculs ou traiter des données.

Avant de pouvoir utiliser un réseau d'ordinateurs pour effectuer ces travaux, il est nécessaire disposer de programmes capable de communiquer ensemble. Dans un réseau comprenant un grand nombre de machines, la configuration manuelle de chacune d'entre elles n'est plus envisageable. La solution trouvée à ce problème est la création d'une dispositif permettant d'installer automatiquement les différents programmes dans ce réseau. Nous appellerons déploiement l'utilisation de ce dispositif. Dans la conception de celui-ci, il faudra tenir compte de l'hétérogénéité des ordinateurs et de leurs contraintes spécifiques.

Enfin, dans le monde business, automatiser le déploiement représente un avantage important car il permet d'installer une application plus rapidement, de manière plus flexible, et d'améliorer la qualité du déploiement en limitant les erreurs de manipulation.

Dans ce travail, nous avons proposé une architecture permettant la réalisation du dispositif cité plus haut. Celui-ci ne peut cependant être utilisé dans tous les cas et il sera donc important de préciser son contexte d'utilisation.

Bien que l'architecture se veut être "générique", certaines applications ne peuvent pas ou sont difficilement déployables parce qu'elles n'ont pas été conçues dans cette optique. C'est le cas par exemple d'applications monolithiques travaillant de manière isolée.

L'architecture présentée ici permet de déployer des applications basées sur des composants. Cette condition est nécessaire car le déploiement d'une application consiste, comme nous le verrons par la suite, à diviser l'application en unité d'exécution indépendantes et à installer ces différents composants sur plusieurs "machines". Cette division ne peut être réalisée que sur des points d'interconnexions bien définis. Cela implique que l'application doit être prédisposée à être déployée.

Nos recherches sur des travaux similaires n'ont abouti qu'à peu de propositions et nous pensons que notre architecture que nous avons validée par l'implémentation réussie d'un prototype représente un progrès dans ce domaine.

Notre travail a enfin comme objectif d'uniformiser les concepts courants dans le monde du déploiement.

Chapitre 2

État de l'art

2.1 Rapide description des systèmes de déploiement existants

Dans les différents articles que nous avons pu lire concernant le déploiement, chaque architecture de déploiement avait ses propres concepts, souvent similaires mais tout de même différents. Ce chapitre décrit rapidement quelques architectures [1] qui nous furent utiles pour la réalisation de la notre. Avant de commencer, nous souhaitons introduire une terminologie afin de clarifier certains termes qui seront utilisés par la suite.

Les systèmes de déploiement utilisent la plupart du temps des composants. Un **composant** est un objet muni des fonctionnalités client/serveur (définit les services dont il a besoin ainsi que les services qu'il peut fournir).

Ces différents composants sont généralement installés sur des **hôtes** ou **noeuds**, c'est à dire tout équipement capable d'exécuter un programme.

Quand ces hôtes sont connectés ensemble, ils forment un réseau et c'est sur lequel on peut distribuer des applications. Un Grid est une réseau dans lequel les ressources d'un grand nombre de machines sont utilisées afin de réaliser des opérations qui nécessitent généralement beaucoup de calculs ou de traitement de grande quantité de données. Les applications traitées dans cette étude possèdent une architecture qui définit la manière dont ses différents composants sont assemblés. Lors d'un déploiement, cette architecture doit être projetée sur l'infrastructure sur laquelle l'application est installée. L'**architecture de déploiement** définit la localisation des différents composants ainsi que les connexions réalisées entre eux.

Nous allons maintenant décrire rapidement quelques architectures.

2.1.1 Architecture permettant des redéploiements fréquents

La première architecture a été décrite par Mikic-Rakic et Medvidovic [7] afin de résoudre certains problèmes comme le déploiement initial d'un système sur un nouvel hôte, le déploiement d'une nouvelle version d'un composant sur un système déjà existant ainsi que l'analyse statique et dynamique des effets des modifications effectuées sur le système.

Les applications déployées sont en réalité des architectures logicielles qui fournissent une abstraction de haut-niveau et qui sont décrites par des **components** chargés des opérations et des états d'un système, des **connecteurs** qui définissent les règles et mécanismes d'interaction entre composants ainsi que les **configurations** qui donnent la topologie des

[1] Une description plus complète des différentes architectures peut être retrouvée dans l'annexe

components et des connecteurs.

La modélisation d'une architecture d'une application peut se faire à deux niveaux : un niveau applicatif, directement en rapport avec les composants, et un niveau meta, qui permet d'observer et/ou faciliter les différents aspects du déploiement, de l'exécution, de l'évolution dynamique ainsi que la mobilité des composants du niveau applicatif.

Il existe deux types de configurations différents pour l'architecture du système. Tout d'abord, une configuration qui décrit la topologie actuelle du système et qui est appelée la **current configuration**. Il existe également la **desired configuration** qui représente la configuration qui doit être déployée. S'il existe une différence entre ces deux configurations, le processus de déploiement est initié.

Ce processus de déploiement est divisé en plusieurs étapes :

1. Un composant de l'architecture de déploiement reçoit tout d'abord la nouvelle configuration désirée. Il l'analyse et si elle est valide, invoque le composant responsable de l'installation des autres composants.
2. Ce composant connait la localisation que doit avoir les différents composants de l'application afin de respecter la nouvelle configuration, il peut dès lors les envoyer sur le réseau.
3. Quand un composant arrive sur un des éléments deployeurs, celui-ci est directement installé et connecté avec les autres composants.
4. Quand ce composant est installé, une notification est envoyée afin de s'assurer que la configuration désirée est bien appliquée.
5. L'application est lancée.

2.1.2 Déploiement sur un Grid

Le déploiement d'application sur un Grid fait intervenir certains concepts un peu différents. Tout d'abord, la complexité d'un déploiement est dûe au grand nombre de noeuds qui peuvent être impliqués. Ces noeuds peuvent être très hétérogènes tant au niveau physique (matériel, réseau, ...) qu'au niveau application (mécanisme de connexion tel que SSH ou de transfert de fichier comme SCP). Les logiciels sont également constitués de composants qui sont hétérogènes en termes de technologies et/ou paradigmes. Un déploiement impliquant autant de noeuds nécessite une grande fiabilité qui n'est possible que par une analyse statique avant l'exécution. Pour terminer, les interconnexions entre Grids créent de nouveaux problèmes, comme les limites des ressources physiques lorsque plusieurs milliers de noeuds sont impliqués.

C'est pour répondre à ces problèmes que les auteurs proposent un framework appelé DeployWare [3], framework qui est constitué d'un DSML [2] afin de modéliser le déploiement et d'une machine virtuelle qui peut interpréter des descriptions et exécuter des processus de déploiement ainsi qu'une bibliothèque de composants de bas-niveau permettant de masquer l'hétérogénéité de l'infrastructure physique.

L'installateur souhaitant réaliser un déploiement doit d'abord créer un modèle de son application qui est compréhensible par l'application. Ce modèle est crée par DeployWare

[2] *Domain-specific modeling language*

sur base d'une description DeployWare qui utilise une syntaxe rigoureuse et qui est réalisée par l'installateur. Cette syntaxe permet de décrire les systèmes à installer et de cacher la complexité. Le modèle est ensuite interprété sous forme d'ensemble de composants qui peuvent être déployés par l'intermédiaire de composants de déploiement.

En ce qui concerne la montée en charge, DeployWare peut être installé comme serveur sur différents noeuds et ainsi répartir le travail de déploiement.

2.1.3 Création d'une architecture de déploiement

Certains problèmes qui n'ont pas été cités dans les architectures précédentes doivent encore être résolus. Lors d'un déploiement, une architecture de déploiement doit être définie pour l'application à être installée. Cette architecture est influencée par un grand nombre de paramètres qui peuvent ne pas être tous connus à la définition du système et qui peuvent varier lors de l'exécution. Enfin, il est difficile de trouver une architecture de déploiement optimale de par le grand nombre de possibilités.

Une méthodologie a été proposée par Malek, Mikic-Rakic et Medvidovic [5] qui permet d'améliorer la disponibilité du système par un monitoring actif du système, une estimation de l'amélioration de l'architecture de déploiement et d'un redéploiement d'une partie ou de la totalité de l'architecture de déploiement. En plus de cette méthodologie, un framework a été développé afin d'adresser les problèmes cités plus haut.

Ce framework est constitué de plusieurs entités qui travaillent ensemble afin de réaliser le déploiement. Il y a tout d'abord un modèle qui maintient une représentation de l'architecture de déploiement du système. Un élément **Algorithm** est chargé de rechercher une architecture afin de satisfaire l'objectif de déploiement. L'architecture décrite par l'**Algorithm** est donnée à un **Analyzer** qui est chargé de la transformer en une séquence d'actions pour satisfaire les objectifs globaux du système. Afin de vérifier les différents paramètres du système, un **Monitor** est attaché à chacune des variables afin de déterminer leurs valeurs à l'exécution. Enfin, l'**Effector** est l'élément qui est chargé d'effectuer le déploiement sur base des instructions de l'**Analyzer**. Il est possible pour l'architecte du système de fournir des paramètres qui ne peuvent pas être facilement mesurables, ainsi que des contraintes sur les architectures possibles.

Ce framework peut être instanciable de manière centralisée où un hôte est défini comme le maître du déploiement. Il est cependant possible d'avoir une instanciation décentralisée du framework où les différents hôtes travaillent ensemble afin de trouver la meilleure architecture possible.

2.1.4 Automatisation du processus de déploiement

Les différentes architectures déjà vues étaient orientées vers la résolution de problèmes de déploiement. Ici, Belguidoum et Dagnat [1] présentent un moyen d'automatiser le processus de déploiement. Pour ce faire, ils ont présenté plusieurs modèles pour le déploiement : les **resources** qui représentent les entités qui sont gérées lors du déploiement, les **mechanisms** qui sont les actions à réaliser pour effectuer le déploiement, les **policies** qui permettent de décrire la manière de choisir les **mechanisms**. Enfin, chacun de ces modèles peut se voir affecter des **properties**.

Ces différents modèles seront utilisés dans une architecture de déploiement. Celle-ci sépare le monde réel du monde formel. Ce monde réel est décrit dans le monde formel afin d'effectuer certaines analyses et raisonnements dans le but de créer des actions abstraites.

Ces actions seront exécutées dans le monde réel en utilisant les mécanismes correspondants afin de réaliser le déploiement. Ces mécanismes représentent les différentes étapes permettant le déploiement concret.

Le déploiement doit être décrit avant d'être exécuté, et les auteurs décrivent comment cela peut être fait. Tout d'abord, les dépendances, qui sont une description des exigences d'une application, sont décrites. Il existe plusieurs types de dépendances : les dépendances obligatoires où une ressource ne peut fonctionner sans la présence d'une autre, les dépendances optionnelles où des services optionnels peuvent être ajoutés à l'application à installer ainsi que des dépendances négatives où une ressource ne peut fonctionner si une autre ressource est présente.

Le contexte est l'ensemble des composants et de leurs relations. La création ainsi que la manipulation d'un tel contexte n'est cependant pas réalisable et une approximation est nécessaire. La solution proposée est constituée d'une liste des composants installés, d'une liste des services fournis et interdits ainsi qu'une liste des composants interdits. Enfin, les dépendances sont représentées par un graphe.

Pour terminer, le moteur de déploiement est l'élément qui représente l'ensemble des règles de déploiement qui peuvent être utilisées par le système. Les phases d'installation et de désinstallation ont été abstraites afin de pouvoir donner des règles permettant de s'assurer de la bonne exécution de ces étapes.

L'étape d'installation est divisée en deux phases. La première phase consiste à vérifier si l'installation est possible, vérification réalisée au moyen du graphe de dépendances ainsi que du contexte. Si l'installation est possible, des règles d'installation sont utilisées afin de calculer les effets de l'installation sur le contexte et des règles permettant sa mise à jour. Cette modification consiste en de nouveaux services disponibles, de nouveaux services interdits, de nouveaux composants interdits ainsi que de nouvelles dépendances entre composants.

La phase de désinstallation est également constituée d'une phase de vérification afin de voir si l'application peut être désinstallée et d'une phase de désinstallation effective avec modification du contexte. Naturellement, des vérifications sont effectuées afin d'empêcher la désinstallation de services nécessaires à d'autres services.

2.1.5 Installation sécurisée de composants

Les architectures précédentes décrivaient principalement comment un déploiement était réalisé ainsi que les différents éléments mis en oeuvre.

Dans cette architecture proposée par Dearle [2], le déploiement tend à être flexible. Cette flexibilité est rendue possible grâce à la possibilité d'installer et d'exécuter du code sur les hôtes distants. Cela va de pair avec des mécanismes de sécurité pour empêcher l'exécution de code malicieux. Le service de déploiement doit également être capable de supporter l'implémentation de composants utilisant des standards de langage de programmation et de modélisations de programmes appropriés ainsi que la possibilité pour les composants de s'interfacer avec des composants déjà déployés.

Cette architecture fournit différents services pour permettre l'installation de composants : mécanismes pour déployer du code et des données, environnement homogène pour les composants déployés, mécanismes de liaisons sûrs, mécanismes pour décrire et déployer des applications distribuées constituées par des composants, la possibilité de faire évoluer

la topologie des applications et des mécanismes de sécurité.

Pour cela, les auteurs ont introduit un serveur appelé **thin server** et qui tourne sur les différents noeuds d'un réseau. Ces serveurs fournissent plusieurs services permettant le déploiement :

- des ports de communication permettant l'envoi de code et de données
- mécanisme d'authentification
- services permettant de stocker des informations sur les composants installés
- un environnement d'exécution appelé **machine**

Pour réaliser une installation, il existe différents outils qui peuvent être envoyés aux différents **thin server**.

Le premier de ces outils est appelé un **installer**. Il permet d'installer un certain nombres de **bundle**. Installer signifie non seulement copier le **bundle** dans la machine, mais également mettre à jour les différentes tables permettant de retrouver ce **bundle**.

Après l'installation, les différents **bundles** sont lancés au moyen d'un **runner**. Cet outil est envoyé sur les **thin server** avec les informations permettant de lancer les **bundles**. Enfin, les différents composants de l'application doivent être connectés afin de réaliser leur travail, c'est le rôle des **wirers**. Ils se connectent aux **thin servers** et créent les liaisons entre les **bundles**.

Voici une description du processus d'installation d'une application : tout d'abord, l'outil de déploiement lit un fichier de description de l'application qui lui permet de connaître des informations nécessaires au déploiement. Il configure alors des **installers** afin d'installer des composants sur différents **thin servers**. Ces **installers** sont alors lancés et renvoient un rapport au moteur de déploiement sur le résultat de cette installation ainsi que des informations pour permettre la suite du déploiement.

Des **runners** sont alors configurés pour lancer les composants nouvellement installés et envoyés sur les différents **thin servers**. Ils renvoient également un rapport au moteur de déploiement avec des informations nécessaires. Enfin, les **wirers** sont configurés et envoyés sur les **thin servers** pour réaliser les connexions entre composants.

Quand toutes les connexions sont réalisées, l'application a été installée et peut effectuer son travail.

2.2 Comparaison de plusieurs architectures de déploiement

La première différence que nous souhaitons détailler concerne la période avant le déploiement. Certaines architectures [7], [5], [1] et [2] nécessitent que des composants soient déjà installés sur les différentes machines afin de permettre le déploiement alors que [3] ne nécessite aucune installation préalable.

Dans le cas où des composants sont déjà installés, il s'agit souvent d'une application qui fait office d'une sorte de machine virtuelle. Les composants fonctionnent alors à l'intérieur de cette architecture. C'est cette machine virtuelle qui permet de déployer les composants sur un grand nombre de noeuds.

Par contre, dans le cas où ces composants ne sont pas présents, il faut utiliser des moyens de distribution fournis par la machine où l'on veut installer les composants (tel

que SSH et FTP), et il est alors nécessaire de fournir les informations de connexion pour les machines impliquées dans le déploiement.

On peut envisager que l'installation du premier type d'outil de déploiement soit faite sur les différentes machines au moyen de ceux du second type.

Une deuxième distinction peut être faite en ce qui concerne le type d'applications qui peuvent être déployées. Il existe des applications qui sont distribuées de par leur conception et d'autres qui doivent le devenir après le déploiement. Dans le premier cas, il faut prévoir le déploiement dès la conception de l'application, et l'application doit donc être adaptée à un tel déploiement. Les outils doivent recevoir des descriptions de la topologie des composants ainsi que des informations sur la manière dont ces composants doivent être instanciés ainsi que la localisation. Dans le second cas, la personne chargée du déploiement ne souhaite généralement pas connaître comment le déploiement sera réalisé. Elle fournit alors l'application ainsi que quelques informations qui permettront de configurer le déploiement. Après le déploiement, il est possible que la localisation des composants soient inconnus par l'installateur.

Les différents outils de déploiement proposent un niveau différent en ce qui concerne le déploiement des applications. Certaines architectures sont assez permissives, dans le sens où elles fournissent principalement des moyens pour installer les différents composants et les connecter sans pour autant fournir des moyens permettant de décider comment ces composants seront installés. C'est le cas par exemple de [2]. Cette architecture fournit des serveurs permettant de faire tourner les composants. Ils possèdent également différents services permettant de connecter les composants, de les lancer et d'effectuer toutes sortes d'opérations sur eux. Nous avons qualifié ce type d'architecture de permissive car il est possible d'installer et d'exécuter tout type de composants désirés ou non. C'est pour cela que des sécurités ont été ajoutées pour empêcher des actes malveillants.

En ce qui concerne les architectures en elles-même, nous avons remarqué qu'un élément était toujours présent : le modèle de l'application à installer. La modélisation de l'application consiste à créer, à partir des informations fournies à l'outil de déploiement, une représentation de l'application de manière standard. Cette modélisation permet par la suite d'effectuer des vérifications et de manipuler un ensemble défini à l'intérieur de l'outil.

Pour terminer, nous avons également voulu faire ressortir le concept de dépendance qui est également présent dans plusieurs architectures. Le fonctionnement d'un composant est dépendant de la présence ou non d'autres composants. Les natures des dépendances sont toutefois assez variées. Celles-ci peuvent être d'un niveau de détail différent, par exemple la nécessité pour un composant de la présence d'une certaine version d'un autre composant. Il existe également des dépendances de plusieurs types, telles que des dépendances obligatoires, optionnelles ou négatives.

Chapitre 3

Proposition d'une nouvelle architecture de déploiement

3.1 Problématique d'une architecture de déploiement

Nous venons de présenter quelques architectures de déploiement ainsi qu'un comparatif des différents concepts qui en ressortent. Avant de proposer notre achitecture, nous souhaitons poser la problématique du déploiement.

3.1.1 Objectifs

Nous allons essayer de cerner les objectifs qu'un déploiement doit rencontrer afin d'être satisfaisant.

Le déploiement doit avant tout permettre de déployer des applications rapidement, à un moindre coût en temps (et en argent) qu'un déploiement manuel et doit être suffisamment flexible pour permettre le déploiement de tous types d'applications. Ceci est d'autant plus vrai dans le monde des entreprises où l'informatique coûte cher et doit poser le moins de problèmes.

Les outils de déploiement sont une solution à ces objectifs puisqu'ils permettent l'installation d'applications sur un grand nombre de "machines", et ce plus rapidement que par l'installation manuelle. La rapidité de mise à jour des différents composants est également un point crucial dans les entreprises puisque celles-ci désirent un temps "d'uptime" le plus élevé possible.

Une architecture de déploiement peut également offrir une plus grande qualité, ce qui permet d'accroître la robustesse de la solution proposée aux utilisateurs.

3.1.2 Contraintes

Un déploiement est toujours réalisé dans un environnement particulier, où les différentes "machines" ont leurs propres contraintes. La difficulté du déploiement vient du fait qu'il doit concilier les objectifs définis par les entreprises avec des contraintes sur les "machines".

Ces contraintes peuvent être très nombreuses car elles touchent tant l'application elle-même qui nécessite un certain contexte pour fonctionner correctement, mais également le déploiement en lui-même car celui-ci est effectué sur des machines qui ont également des contraintes.

Le nombre de "machines" disponibles pose également un problème supplémentaire. Dans le cas où il est nécessaire d'installer une application et que ces "machines" ont une architecture identique, cette installation peut être envisagée en installant une des machines

et en copiant sa configuration sur les autres machines. Cette solution n'est plus envisageable dans un contexte où les machines peuvent avoir des configurations très différentes. Il est en effet nécessaire d'installer chaque application de manière personnalisée.

L'utilisation d'outil de déploiement est donc justifié dans le sens où les administrateurs ne doivent plus s'occuper de la configuration des différentes applications mais de la description de celles-ci.

Dans la suite de ce travail, nous allons proposer notre architecture de déploiement. Il est nécessaire d'introduire certains concepts afin de pouvoir mieux comprendre le fonctionnement de celle-ci. Nous justifierons alors nos différents choix ainsi qu'une critique de ceux-ci.

3.2 Description des concepts utilisés dans l'architecture

Avant d'introduire notre proposition d'architecture, il convient de présenter certains concepts permettant une meilleure compréhension.

3.2.1 Candidature

Une candidature est une liste des composants qu'un Deployer peut installer. Cette liste est envoyée au Master Deployer qui pourra alors créer le plan de déploiement.

3.2.2 Composant

Un composant est la plus petite unité qui existe lors d'un déploiement. Une application est le résultat de la combinaison de composants liés entre eux par une architecture [1].

3.2.3 Déploiement

Un déploiement consiste en l'installation, la configuration et l'utilisation d'une application. Le déploiement dans cette architecture permet de transformer une application basée sur des composants en une application distribuée sur un ensemble de noeuds. Cela est possible par la description des interactions entre les différents composants.

3.2.4 Deployer

Le Deployer est l'entité chargée de déployer les composants d'une application.

3.2.5 Master Deployer

Le Master Deployer est le Deployer à qui la demande d'installation est faite. C'est lui qui est alors chargée de coordonner le déploiement.

3.2.6 Naming Service

Le Naming Service est une application qui tourne sur un des Deployers et qui contient une table permettant de récupérer un composant sur base de son nom. Cela permet de connecter des composants qui sont déployés sur plusieurs Deployers.

[1] Cette architecture est décrite par un ADL : Architecture Description Language. Il s'agit de notations formelles de modélisation permettant de décrire des architectures. Ces langages sont généralement supportés par des outils de développement. Les ADL permettent de se concentrer sur la structure de l'application à un haut-niveau plutôt que de se préoccuper des détails d'implémentation.

3.2.7 Package

Le **Package** représente l'application qui doit être installée, configurée, ... par l'application de déploiement. Ce package doit contenir l'application elle-même mais également les informations afin de pouvoir être installée lors du déploiement (le fichier *package.xml*).

3.2.8 Plan de déploiement

Le plan de déploiement est l'architecture de déploiement de l'application à installer. Ce plan assigne un Deployer à chaque composant. Ces composants seront en réalité installés sur les Mini Deployers. Ce plan est créé par le Master Deployer lors de l'instanciation de l'application.

3.3 Description de l'architecture de déploiement proposées

Dans ce chapitre, nous allons décrire l'architecture de déploiement que nous proposons. Celle-ci est basée sur Fractal [2] qui est un modèle de composants.

L'architecture a été divisée en deux parties : Le **Deployer**, responsable de la réception des packages à installer, des vérifications avant le déploiement, ...Il s'agit de la partie qui est en charge du déploiement en général. Le **Mini Deployer** est la partie qui est installée sur les différents noeuds et est responsable de faire tourner les composants des applications.

Ces deux parties interagissent ensemble pour réaliser le déploiement de l'application. Nous allons donc tout d'abord présenter la vue générale d'un déploiement avant de décrire ces deux éléments dans le détail.

3.3.1 Description de l'architecture en général

Le déploiement d'une application implique certaines étapes : demande d'installation d'une application, modélisation de cette application et installation. Ces étapes peuvent être résumées comme présenté à la figure 3.1.

La première étape, appelée **load**, est la demande d'un client pour le chargement d'un package. C'est cette étape qui lance le déploiement. Le Deployer effectue l'extraction du package à l'étape **extract** et récupère les informations afin de déployer l'application. L'étape suivante **sendPackage** consiste en l'envoi du package aux Deployers afin qu'ils puissent également avoir les informations concernant le package à installer. L'envoi du package lance un processus similaire à l'étape **load** du client. L'étape **createModel** permet de créer le modèle de l'application à installer. Ce modèle est vérifié à l'étape **validation**. Lors de cette validation, les Deployers créent une liste avec les différents composants qu'ils sont capables d'installer. Ils envoient alors cette liste sous forme de candidature au premier Deployer. Il s'agit de l'étape de **candidature**.

Quand le premier Deployer a reçu toute les candidatures, il peut lancer l'étape **instantiate** qui démarre la création des composants, leurs liaisons ainsi que toutes les autres opérations nécessaires à l'installation et à l'exécution de l'application. À chaque demande d'instanciation d'un composant sur un Deployer, celui-ci ordonne à un de ses Mini Deployers d'installer le composant à l'étape **install**.

Enfin, quand tous les composants sont installés et connectés entre eux, le déploiement en tant que tel est terminé. Il ne reste plus que l'étape **launch** qui lance l'exécution de l'application.

[2] Pour plus de renseignement concernant Fractal, voir annexe H

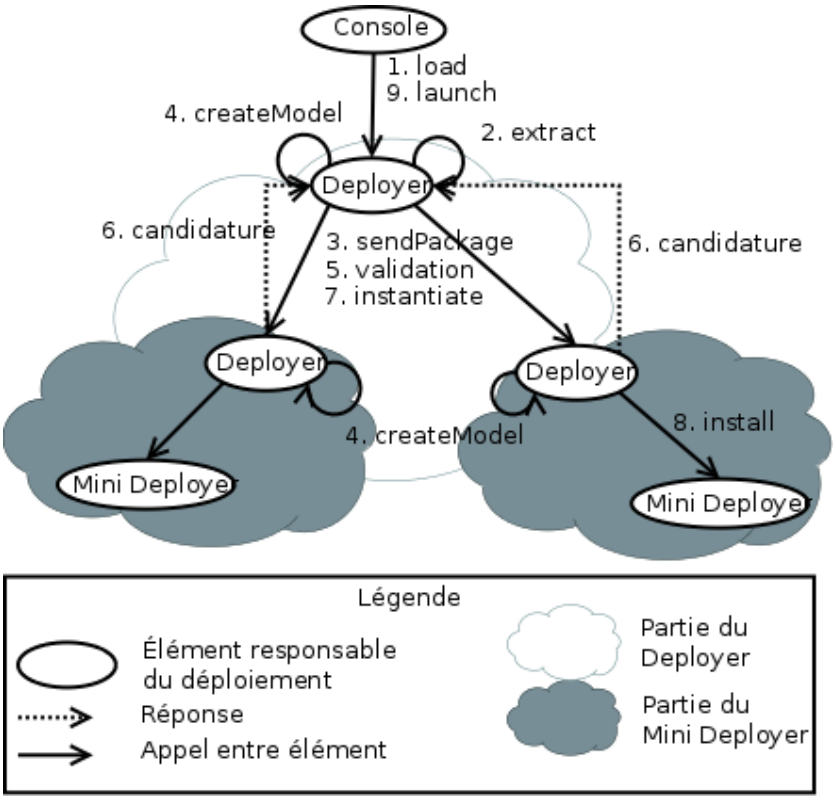


FIGURE 3.1 – Architecture générale décrivant les interactions entre Deployer et Mini Deployer

3.3.2 Architecture du Deployer

Le **Deployer** est la partie responsable du déploiement d'une application. C'est dans cette partie que l'intelligence de l'outil de déploiement se trouve. Une représentation graphique du **Deployer** est présenté à la figure 3.2. Celle-ci utilise la formalisation graphique de Fractal : Les rectangles représentent les composants de l'application Fractal. Les éléments situés sur les côtés gauches et droits des composants sont appelés les interfaces et définissent les points d'accès aux composants. Les liaisons entre les composants sont représentés par les traits reliant les interfaces. Enfin, les éléments situés au dessus des composants sont appelés des controllers mais ils ne sont pas utilisés dans notre architecture.

Nous allons maintenant décrire chacun des composants de l'architecture dans l'ordre dans lesquels ils sont généralement utilisés.

Le **Controller** est le composant qui permet de contrôler l'installation d'un package. C'est le point central de l'architecture car il agit comme un chef d'orchestre en faisant interagir les autres composants. Ce composant est le plus important car c'est à travers lui que toutes les communications sont réalisées et que le déploiement est réalisé. Il a accès à pratiquement l'entièreté des composants.

Ce composant interagit avec les autres composants au moyen de deux points d'entrées. L'interface **cmd** est utilisée par un client pour effectuer des opérations telles que des installations, mises à jour, ... De cette manière, le client ne peut interférer avec la partie responsable du déploiement. L'interface **communication** s'occupe de la communication entre les Deployers d'un part et entre les Deployers et leurs Mini Deployers d'autre part.

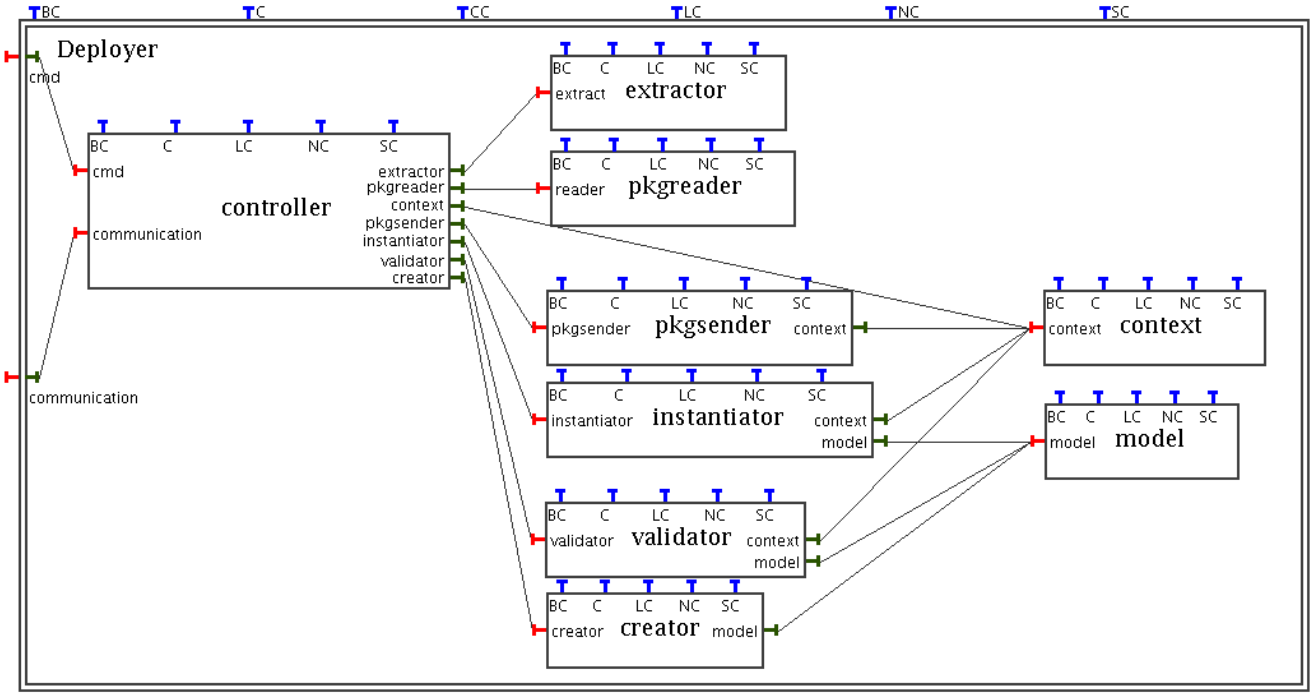


FIGURE 3.2 – Représentation graphique de l'architecture du Deployer

Le composant suivant est l'**Extractor** qui est chargé d'extraire les fichiers contenus dans le package fourni par le client. Il est également chargé d'abstraire l'accès aux fichiers du package pour les autres composants.

Après avoir récupéré le contenu du package, le **Pkgreader** (ou **Package Reader**) est chargé de lire la description du package à installer afin d'avoir des informations nécessaires pour le déploiement de l'application. Ces informations sont :

- Le nom du package.
- Le type du package : Fractal, OSGi, ...
- Le fichier décrivant l'application à installer suivant le type de package.
- Le point de lancement de l'application.
- Les contraintes sur les composants. Il peut s'agir de contraintes concernant des aspects matériels (espace mémoire nécessaire, débit réseau, ...), des aspects logiciels (version du système d'exploitation, présence de certains composants, ...) ou d'autres types de contraintes comme la localisation de certains composants.

Plusieurs autres composants se servent des informations du package afin d'effectuer leur travail. L'**Instantiator** utilise les contraintes afin de définir l'architecture de déploiement ainsi que le fichier décrivant l'architecture de l'application. Le **Controller** récupère le type du package afin d'adapter les différents composants pour le déploiement ainsi que le point de lancement de l'application.

Le composant **Pkgsender** est chargé d'envoyer le package aux autres Deployers. Pour cela, il récupère la liste des Deployers connectés au système conservée par le **Context** . C'est le **Controller** qui demande à ce composant d'envoyer le package.

Le **Context** est un composant qui représente l'état du système à tout moment. Il doit tout d'abord contenir des informations sur les noeuds où les composants sont installés, sur les services fournis par les composants ainsi que les applications qui nécessitent ces composants. Il contient également des informations sur les Deployers qui sont connectés sur le système. Ce composant est utilisé par l'**Instantiator** lors de l'instanciation des composants et par le **Validator** pour effectuer certaines vérifications.

Le **Creator** est le composant qui se charge alors de créer un modèle de l'application à installer. Ce modèle permet d'effectuer des opérations comme des vérifications, des analyses, ... Ce composant est activé par le **Controller** . La création du modèle est laissé à la charge du composant et dépend du modèle de composants utilisé par l'application à installer. Il communique avec le **Controller** via son interface **creator** et peut créer le modèle de l'application en utilisant son interface **model**.

Le **Model** n'est pas un composant comme les autres dans le sens où ce n'est pas un composant fonctionnel. Il ne sert qu'à représenter le modèle de l'application à installer. Il contient typiquement les différents composants de l'application, les liaisons entre eux ainsi que toutes autres informations nécessaires à l'installation de l'application.

Le contenu de se composant est tout d'abord alimenté par le **Creator** au début de l'installation. Le **Validator** l'utilise afin de vérifier si l'installation est possible ainsi que d'autres vérifications et analyses. Enfin, l'**Instantiator** récupère les différents composants de l'application afin de les instancier dans le système, les liaisons à effectuer entre eux ainsi que les autres informations nécessaires pour accomplir le déploiement.

Le **Validator** est le composant chargé de la vérification du modèle de l'application à installer. En effet, le modèle ne peut comporter d'erreurs, de références à des composants inexistants, ... De plus, le **Validator** doit vérifier que certaines conditions soient respectées comme par exemple un nom unique pour chaque composant afin de pouvoir les identifier. Lors de la validation du modèle, ce composant crée une liste qui contient l'ensemble des composants que le Deployer est capable d'installer.

Enfin, l'**Instantiator** est le composant chargé de la création et de la mise en fonction des composants à déployer. Cette étape est nécessaire afin de pouvoir utiliser l'application déployée. Ce composant possède deux comportements : le **Master**, qui s'occupe de l'instanciation des composants sur les autres Deployers. C'est également lui qui est chargé de lancer le *Naming Service* qui sera utilisé par l'application. Le deuxième comportement, appelé **Slave**, se charge simplement de la réception et de l'exécution des demandes d'instanciations. Les composants ne sont pas instanciés directement sur les Deployers, mais sur les Mini Deployers et ceci afin de séparer l'outil de déploiement de l'utilisation des composants dans les applications. Le choix du Mini Deployer sur lequel le composant sera déployé est laissé à l'appréciation du Deployer.

3.3.3 Architecture du Mini Deployer

La représentation graphique du Mini Deployer est présentée à la figure 3.3. Le Mini Deployer est chargé de la création et de l'exécution des différents composants de l'application à la demande d'un **Instantiator** .

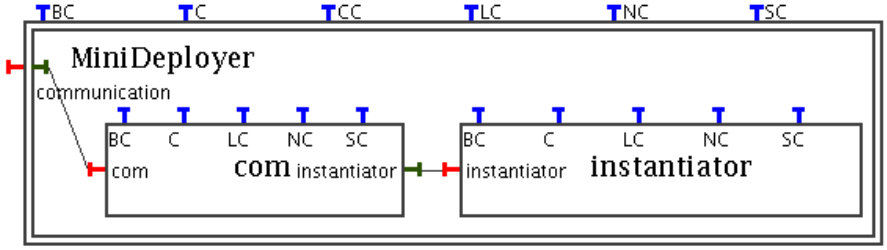


FIGURE 3.3 – Représentation graphique du Mini Deployer

L'architecture du Mini Deployer est beaucoup plus simple afin de rendre cette partie la plus petite possible.

Le premier composant est celui chargé de la communication avec le Deployer et est nommé le **com** ou **Mini Communication**. C'est par son intermédiaire que le Mini Deployer est notifié quand il doit installer un composant et qu'il peut notifier au Deployer que le composant est correctement installé.

L'**Instantiator** ou **Mini Instantiator** pour ne pas le confondre avec le composant du Deployer est le composant qui est chargé de l'instanciation des composants. Cette étape est dépendante de facteurs tels que le type du composant ou du système sur lequel le composant est instancié.

3.4 Justification des choix de l'architecture

Dans ce chapitre, nous allons détailler les choix que nous avons fais concernant l'architecture telle que présentée.

3.4.1 Séparation Deployer / Mini Deployer

L'architecture proposée fait une distinction entre le **Deployer** et le **Mini Deployer**. Par cette séparation, nous voulions séparer la partie responsable du déploiement de celle responsable de faire tourner l'application déployée. Cette séparation est justifiée si l'on veut pouvoir couper l'application de déploiement. Un déploiement peut alors être réalisé comme ceci :

- Lancement de l'application de déploiement.
- Lancement des Mini Deployers.
- Installation de l'application.
- Lancement de l'application déployée.
- Arrêt de l'application de déploiement.

De cette manière, les Deployers ne sont pas nécessaires pour que l'application puisse fonctionner [3].

[3] Ce n'est actuellement pas le cas dans le prototype.

Cette séparation est d'autant plus justifiée que le **Deployer** est une grosse application qui nécessite beaucoup de ressources. Nous trouvions donc inutile d'utiliser autant de ressources pour au final n'utiliser qu'un seul composant. Le **Mini Deployer** a donc une architecture très simple et qui nécessite peu de code pour faire tourner un composant. Nous avions également envisagé que le Mini Deployer puisse être installé sur des téléphones mobiles ou tout du moins sur des périphériques ne possédant que peu de ressources matérielles. L'idée d'un **Mini Deployer** provient de l'article [5]. Le Mini Deployer est en effet une version très simplifiée du Deployer puisqu'il ne possède qu'un composant de communication et un composant d'instanciation de composant.

3.4.2 Controller

Pour rappel, ce composant est responsable de l'ordonnancement des opérations de déploiement. Ce composant permet également de coordonner la communication entre les Deployers pour la réception des candidatures, l'envoi des packages et autres informations. C'est également ce composant qui communique avec les Mini Deployers.

3.4.3 Model

Nous trouvons que la création d'un modèle de l'application à installer est un point particulièrement important dans le déploiement, c'est pour cela que ce composant fut un des premiers présents dans l'architecture. Un modèle permet d'abstraire certains concepts pour ne garder que les parties qui sont nécessaires au déploiement. Quand le modèle est créé, il est possible d'effectuer certaines opérations sur celui-ci :

- Vérification : il est en effet plus facile d'effectuer des vérifications sur un objet connu que sur des descriptions de composants, de fichiers de configuration, ...
- validation : il est possible de valider le modèle pour voir s'il respecte certaines exigences.

Nous avions aussi espéré pouvoir créer un modèle unique pour tout déploiement et adapter les autres composants pour s'adapter à ce modèle. En effet, si le modèle ne change pas, certains composants qui travaillent uniquement sur le modèle ne doivent pas être modifiés (ce qui est le cas du **Validator** par exemple).

Par exemple, en Fractal, il existe des composants composites et primitifs alors qu'en OSGi il n'existe pas de composants composites. Un **Creator** pour OSGi ne créerait alors que des composants primitifs sans que cela n'affecte le modèle. Il s'agit cependant d'une zone à explorer.

3.4.4 Extractor, Pkgsender et Pkgreader

Ces composants sont d'une utilité moindre que les précédents. Ces composants servaient au début à simplifier le développement du prototype. L'**Extractor** fournit un accès facile aux fichiers du package à installer. Le **Pkgreader** permet également de simplifier la lecture des informations du package et **Pkgsender** simplifie l'envoi du package aux autres Deployers.

Cependant, grâce au modèle de composants et aux ADL (Fractal ici), il est possible d'avoir une autre implémentation de ces composants et ce facilement puisqu'il suffit de modifier le fichier ADL pour avoir une autre implémentation. De ce fait, les choix que nous avons réalisés comme un fichier *zip* pour le package, un fichier *package.xml* qui contient les informations du package, ... peuvent être adaptés facilement puisqu'il suffit de respecter l'interface. Donc, au lieu d'avoir un **Extractor** pour les fichiers *zip*, un **Pkgreader** pour le fichier *package.xml*, il est possible de travailler à un niveau plus abstrait et de parler d'une archive, d'un fichier de description, ... et ceci peut simplifier grandement l'implémentation de cette architecture.

3.4.5 Fractal comme modèle de composants

Le prototype ainsi que l'architecture repose beaucoup sur le modèle de composants de Fractal.

Nous avons choisi Fractal pour ses nombreuses qualités :

- Il est très bien adapté au déploiement et à la reconfiguration d'applications
- Il est modulaire et relativement simple d'apprentissage.
- Il est régulièrement mis à jour et est open-source
- Il existe de nombreux projets basés sur Fractal et que nous avons utilisés :
  - Fractal pour Eclipse.
  - Fractal GUI qui permet de décrire l'architecture d'une application de manière graphique.
  - Fractal ADL qui permet de décrire l'architecture de l'application en utilisant des fichiers xml.
  - Les nombreuses implémentations de la spécification en C, .NET, ... qui permettent théoriquement d'utiliser l'architecture dans ces langages.

3.4.6 Validator

Ce composant est probablement un des plus importants si un des objectifs principal est d'avoir une architecture de déploiement optimale. Dans l'état de l'art, nous avons décrit plusieurs architectures, mais peu d'entre-elles s'intéressaient réellement à cet aspect. L'article [5] discutait justement des différentes stratégies afin de trouver une architecture optimale.

Ici, la validation n'est qu'une partie de cette recherche d'une architecture car elle vérifie simplement quels composants peuvent être installés par le Deployer.

3.4.7 Instantiator

La création des composants est l'objectif du déploiement, un composant de l'architecture de déploiement lui est donc naturellement dédié.

Chapitre 4

Implémentation de l'architecture dans un prototype

4.1 Construction du prototype

Dans cette section, nous allons présenter les étapes de développement du prototype. Ce prototype utilise le modèle tel que présenté au chapitre 3.3.

Nous tenons tout d'abord à préciser que le développement de l'architecture et du prototype ont été réalisés de manière parallèle. Nous avons choisi Fractal comme base pour le développement du prototype pour les raisons déjà précisées dans la justification des choix de l'architecture.

Définitions de base

La première étape a été de définir quelles étaient les entités participant au déploiement avec les services qu'elles pouvaient fournir et qu'elles exigeaient, leurs rôles (fonctionnalités) ainsi que leur connexion.

Nous nous sommes alors interrogé sur les éléments qui sont importants lors d'un déploiement. Sur base des articles, nous avons vu que certains concepts devaient nécessairement être représentés dans une application de déploiement tandis que d'autres ressortaient plus de cas particuliers. Nous étions très intéressé par le concept de modèle de l'application (d'ailleurs présent dans les différents articles sur le sujet) car la modélisation permet de ne garder que les *concepts* importants de l'application. Afin de disposer de ce modèle, il était nécessaire d'avoir un composant chargé de sa création. Le modèle une fois créé, nous l'instancierons lors du déploiement.

Premier prototype

C'est sur ces idées que nous avons créé le premier prototype. Il était composé de 5 composants (qui sont restés dans l'architecture finale) : le controller chargé de contrôler le déploiement et de la communication avec l'extérieur, l'extractor pour extraire les fichiers de l'application à installer, le créateur, chargé de créer un modèle de l'application à déployer et l'instanciator qui instancie les composants. Comme les premières applications devant être déployées utilisaient également Fractal, l'instanciator était spécifique à Fractal et possédait des interfaces permettant la création des éléments spécifiques à Fractal. Le modèle devait être générique (puisque l'architecture devait être générique) afin d'avoir une partie fixe présente dans le prototype. Le créateur devait alors être adapté ainsi que l'instantiator pour créer le modèle.

Le modèle représentait parfaitement les applications Fractals, mais n'étaient pas générique et ne pouvaient donc par représenter d'autres modèles tels que OSGi. Nous avons

alors conçu le prototype afin qu'un creator pour OSGi crée un modèle compatible avec le modèle du prototype. Cette solution n'est cependant pas intéressante car il existe des incompatibilités entre les deux modèles.

Nous avons alors décidé que le creator, le modèle et l'instanciateur devaient être spécifique à l'application qui devait être installée.

"Proof of concept"

A ce niveau du développement, il n'était pas encore possible de déployer une application. Afin de simplifier le déploiement, nous avons fixé les différentes étapes pour la réalisation du prototype.

Il fallait donc un "proof of concept" afin de détecter d'éventuels problèmes. L'outil de déploiement devait alors pouvoir effectuer une simulation de déploiement, mais dans un cadre centralisé. Les différents composants impliqués dans le déploiement étaient simplement chargés d'afficher des messages informant la bonne réalisation du déploiement.

Première application Fractal déployée

Quand cette partie fut fonctionnelle, il était temps de passer à l'installation d'une application Fractal. Nous avons rencontré un obstacle car il était nécessaire de créer un outil permettant de créer un modèle à partir des fichiers de descriptions ADL de l'application à installer. Ces fichiers de descriptions constituent une couche supérieure aux applications Fractals et nous devions implémenter nous-même certains concepts.

Nous nous sommes contenté d'implémenter un sous-ensemble des fonctionnalités, reprenant celles dont nous avions besoin.

Déploiement distribué

Nous avons alors commencé à réfléchir au déploiement distribué. Pour cela, il était nécessaire d'adapter l'architecture pour ajouter un moyen de communication entre les Deployers mais également d'adapter l'instanciator afin que les composants puissent être connectés de manière distribuée. Après quelques recherches, nous avons trouvé que Fractal proposait une bibliothèque, appelée Fractal RMI, et qui possédait toutes les exigences voulues.

L'ajout de cette bibliothèque fut très rapide et le prototype a rapidement pu déployer une application Fractal de manière distribuée.

OSGi

Après Fractal, nous nous sommes intéressé à étendre les modèles de composants pouvant être déployés par notre application. Nous avons choisi OSGi. Dans ce modèle, les composants sont appelés des *bundles* mais sont beaucoup plus autonomes. Nous n'avons cependant pas pu trouver d'équivalent à Fractal RMI pour OSGi.

Nous avons alors commencé à développer des *bundles* qui pourraient être installés dans les plate-formes OSGi et qui pourraient effectuer le travail de distribution.

4.2 Description du fonctionnement du prototype

Cette section va décrire le principe de fonctionnement du prototype. En ce qui concerne les aspects techniques, ceux ci peuvent être trouvé dans l'annexe A.

L'installation d'un package peut être divisée en plusieurs étapes :

1. Demande du client à un Deployer d'installer un certain package.

2. Création par le Deployer d'un modèle représentant l'application à installer.

3. Validation du modèle afin de savoir s'il est possible de l'installer.

4. Lancement de l'installation du package.

5. Lancement de l'application contenue dans le package.

Demande d'installation du client

Le client se connecte tout d'abord sur le Naming Service afin de récupérer la référence au Deployer sur lequel il doit installer le package (1). Il peut alors se connecter sur l'interface de communication du Deployer et lancer la procédure d'installation du package (2).

Cette étape est décrite par le diagramme de séquence de la figure 4.1.

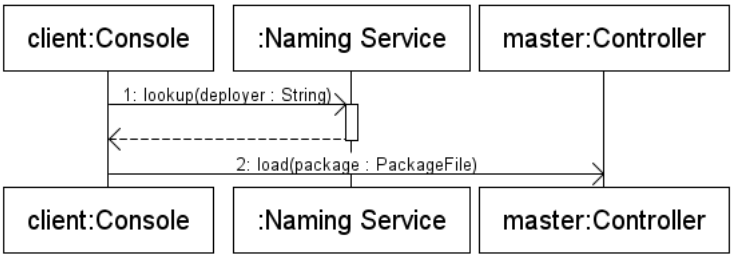


FIGURE 4.1 – Demande d'installation

Création du modèle

Lors d'une demande d'installation d'un package, le Deployer fait extraire les fichiers du package par l'**Extractor** dans un dossier temporaire (1). Il demande ensuite au **Pkgsender** d'envoyer le package aux autres Deployers afin qu'ils puissent l'extraire également (2).

Le **Controller** demande au composant **Pkgreader** de lire le contenu du fichier de description. Pour rappel, la description du package donne des informations sur le package à installer comme le nom de l'application, le type de modèle utilisé, ... Le **Pkgreader** retourne un objet **Package** au Deployer. En utilisant ce **Package**, le **Controller** demande au **Creator** de créer le modèle de l'application à installer (5). Le **Creator** peut alors créer un nouveau **Model** (6). En Fractal, ce fichier de description est celui du composant principal. La création du modèle est naturellement adapté au type de modèle du package.

Dans le prototype, le seul modèle reconnu est le modèle de composant Fractal. Afin de créer le modèle, le **Creator** récupère tout d'abord la liste des fichiers de descriptions Fractal (qui sont en fait des fichiers xml). Ils les parsent afin d'en retirer les informations nécessaires pour créer le modèle du composant. Ces modèles sont alors rassemblés dans le **Model** et c'est cet ensemble qui forme le modèle du package à installer.

Cette étape est décrite par le diagramme de séquence de la figure 4.2.

Validation du modèle

Tous les Deployers demandent à leur **Validator** de valider le modèle (1). La validation permet de s'assurer que l'application à installer ne comporte pas d'erreurs (composants manquants, liaisons impossibles, ...), mais également à vérifier que les contraintes (comme la mémoire vive nécessaire, ...) sont respectées. Après la validation, le **Validator** possède

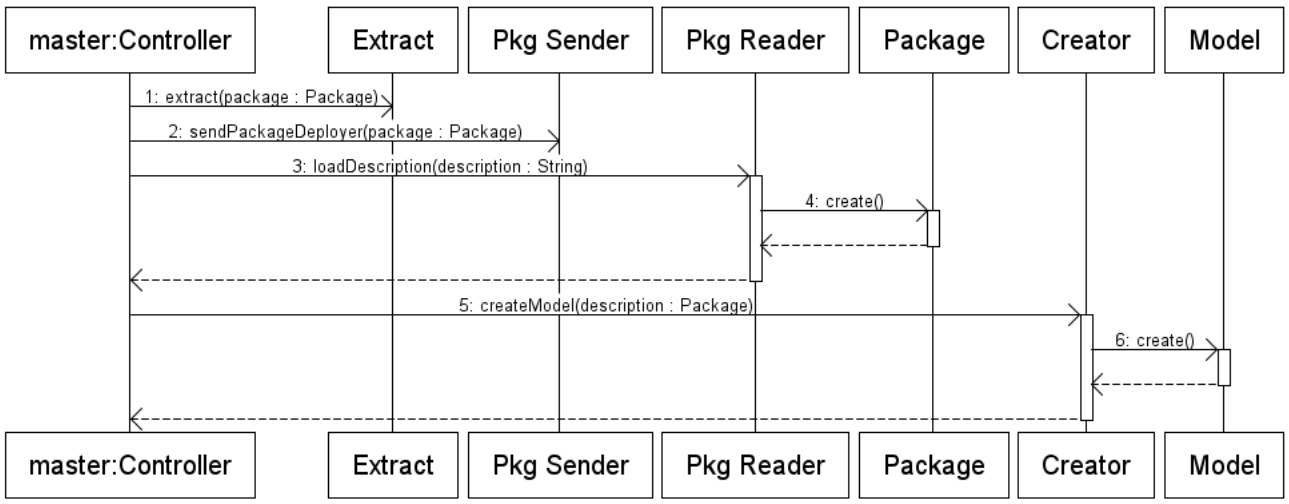


FIGURE 4.2 – Création du modèle

une liste des composants que le Deployer peut installer. Les Deployers autre que le master envoient alors la liste des composants qu'ils peuvent installer au master (2). Cette liste de composants est appelée la **candidature** du Deployer.

Cette étape est décrite par le diagramme de séquence de la figure 4.3.

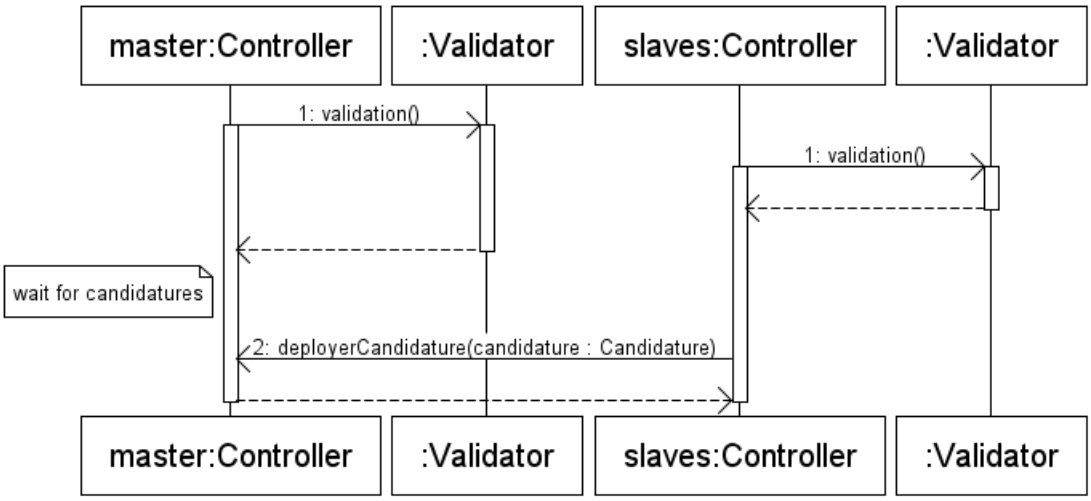


FIGURE 4.3 – Validation du modèle

Installation du package

Le Master Deployer demande à son composant **Instantiator** d'installer le composant principal (1). L'**Instantiator** commence tout d'abord par créer un plan de déploiement qui assigne un Deployer à chaque composant (2) et l'envoie aux différents Deployers (3). L'élaboration du plan de déploiement est un point critique du déploiement et doit être optimale. Dans le prototype, la création du plan de déploiement est relativement simple et essaie d'assigner des composants sur chaque déployer.

L'**Instantiator** récupère enfin le Deployer sur lequel le composant principal doit être installé et envoie une demande à ce Deployer pour installer le composant (4). Le De-

ployer concerné lance tout d'abord un nouveau Naming Service afin d'enregistrer tous les composants de l'application (5).

Comme le composant principal est un composant composite, il faut d'abord que tous les sous-composants soient installés et connectés entre eux. Pour cela, le Deployer récupère le Deployer sur lequel chaque sous-composant direct doit être installé et envoie une demande d'installation de ce sous-composant à ce Deployer (6).

Étant donné que les composants ne sont pas réellement installés sur les Deployers mais sur les Mini Deployers, il faut donc que le Deployer transmette la demande d'installation du composant à un Mini Deployer (7).

Cette étape est décrite par le diagramme de séquence de la figure 4.4.

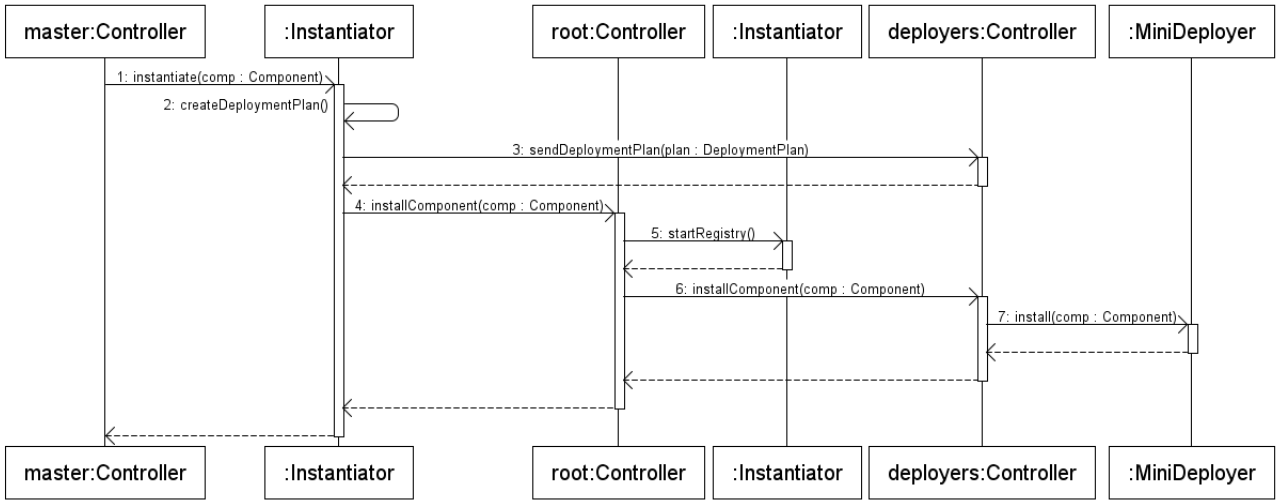


FIGURE 4.4 – Installation du package

Lancement de l'application

Quand tous les composants sont installés, il ne reste plus qu'à lancer l'application. Pour cela, le **Controller** récupère le Naming Service de l'application, récupère le composant principal (1) et le lance en utilisant les informations contenues dans le *package.xml* (2).

Cette étape est décrite par le diagramme de séquence de la figure 4.5

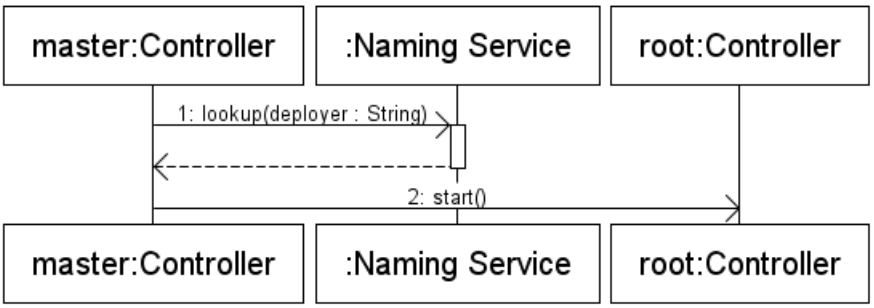


FIGURE 4.5 – Lancement de l'application

4.3 Limites du prototype

Le but du prototype était de créer une application qui permettait le déploiement d'applications et qui utilise l'architecture définie auparavant.

Le prototype ne permet actuellement que de lancer le déploiement d'une application utilisant le modèle de composant Fractal. Il était en effet plus facile de créer l'application de déploiement ainsi que des applications d'exemple utilisant le même modèle. De plus, Fractal permet de créer une application distribuée de manière transparente pour l'application.

Afin de déployer une application Fractal, il faut tout d'abord analyser l'architecture de l'application à déployer. Cette analyse est nécessaire afin d'avoir une vue d'ensemble de l'application. Il n'était malheureusement pas possible d'utiliser directement la bibliothèque Fractal pour réaliser cette opération, nous avons donc dû implémenter un parseur pour effectuer ce travail. Étant donné qu'il s'agissait d'un prototype, ce parseur n'est pas aussi évolué que celui de Fractal et ne permet pas de réaliser toutes les possibilités. Notre implémentation est en fait un sous-ensemble des fonctionnalités proposées par l'implémentation officielle. Cependant, l'important était d'élaborer une architecture de déploiement et non un prototype, cette limite ne doit pas porter à conséquence.

Le prototype ne peut installer qu'un seul package. Cette limitation empêche de pouvoir effectuer des installations utiles comme installer simplement un seul composant. Il n'est donc pas possible d'utiliser un composant déjà installé. Ces limitations ne sont pas très importantes et devraient facilement être dépassées dans le cadre d'une amélioration du prototype. Il ne s'agit pas d'une limitation de l'architecture en elle-même.

La seule opération possible sur le prototype est l'installation d'un package. Il n'est pas possible d'effectuer de désinstallation ou de mise-à-jour de composants ou de l'architecture d'une application déjà installée. Ces opérations nécessitent un plus grand travail.

Enfin, ce prototype transforme une application à la base locale en une application distribuée. Il n'est pas possible d'installer une application qui est déjà distribuée. On ap-

pelle une application distribuée une application qui utilise déjà Fractal RMI. En effet, une application distribuée nécessite des opérations supplémentaires qui ne sont pour l'instant pas intégrables dans le prototype comme par exemple l'utilisation d'un naming service, la définition de la localisation des composants, ... Le prototype choisit arbitrairement ces informations (emplacement du Naming Service et des composants, ...)

Chapitre 5

Critiques

Après avoir décrit notre architecture de déploiement, il convient de réaliser une critique de celle-ci.

5.1 Points forts de l'architecture

Une des qualités de l'architecture proposée, c'est qu'elle est modulaire. Cela signifie qu'il est possible de modifier rapidement certains des composants sans affecter les autres composants. Il est par exemple envisageable de transformer le composant **Extractor** afin que les fichiers ne soient pas extraits en local mais sur un serveur global.

Un autre avantage est la présence du composant **Model** qui permet d'avoir une abstraction de l'application à installer. Cette abstraction est nécessaire pour permettre tout d'abord de cacher la complexité de certains éléments de l'architecture mais également de permettre une manipulation uniforme. Ainsi par exemple, les tests de validation et de vérification peuvent être identiques pour deux implémentions de l'architecture.

Enfin, cette architecture est utilisée actuellement dans le prototype qui peut déployer des applications. Cela signifie que cette architecture est donc capable de gérer le déploiement d'applications. Le prototype transforme l'application en application distribuée, et ce, de manière transparente pour l'application.

5.2 Opportunités d'améliorations

Comme vu précédemment, Fractal est utilisé pour réaliser le déploiement des applications. Cette bibliothèque ne pose aucun problème lors du déploiement d'applications Fractal. Mais qu'en est-il des autres modèles de composants ? Fractal est-il adapté pour le déploiement d'autres modèles comme OSGi ? Nous avons essayé de nous pencher sur le déploiement de bundles OSGi. Alors qu'il est facile de rendre une application distribuée grâce à Fractal, ce n'est pas le cas en OSGi. Fractal RMI, bibliothèque utilisée pour le déploiement distribué, simplifie grandement l'implémentation. Pour OSGi, il est nécessaire de créer un bundle qui se charge de rendre transparent le cadre distribué de l'application. Il se trouve qu'il existe peu de solutions permettant d'interconnecter plusieurs plate-formes OSGi.

L'architecture proposée manque d'un élément qui peut avoir de l'importance : la sécurité. Rien ne prouve en effet que les différents composants qui vont s'installer sur les différents noeuds ne sont pas malveillants. Il est cependant intéressant de se demander si c'est l'architecture qui doit gérer cette situation ou si c'est à l'outil de déploiement. Dans l'article [2], un certificat est attaché à chaque bundle, ce qui permet de s'assurer que celui-

ci est bien celui qu'il prétend être. Cela peut être une voie vers une solution. De manière générale, il manque la possibilité de définir des exigences (la sécurité par exemple) afin d'influencer le déploiement.

Le prototype ne montre que des exemples de déploiement en Fractal alors qu'il est censé implémenter une architecture permettant un déploiement générique. De plus, ces exemples sont des *proof of concepts* et non des applications industrielles.

Chapitre 6

Développement futur

Après avoir présenté l'architecture de déploiement, quelles sont les perspectives de travail pour le futur ? Voici quelques pistes de réflexions :

Autres modèles de composants

Actuellement, le prototype ne permet que de déployer des applications Fractal. Une première modification intéressante serait l'intégration d'autres modèles de composants. Il est en effet difficile de valider une architecture générique alors qu'un prototype ne permet pas de l'être.

Utilisation des "Package Managers"

Les "Package Managers" ou les gestionnaires de packages sont des applications utilisées dans les distributions Linux et qui permettent d'installer des applications. Ces gestionnaires de packages ont un comportement semblable aux applications de déploiement à plusieurs niveaux :

- installation d'application.
- gestion des dépendances.
- ...

Nous pensons qu'il pourrait être intéressant d'intégrer ces "Package Managers" dans une architecture de déploiement.

Montée en puissance

Le prototype a été développé afin de montrer qu'il était possible d'implémenter l'architecture présentée. Pour tester ce prototype, nous avons créé quelques exemples. Ces exemples ne sont constitués que de quelques composants (pas plus de 5). L'exemple le plus important nécessite 5 composants, 3 Deployers et 3 Mini Deployers. Avec le Naming Service et le client qui lance l'installation, cela fait 8 applications à lancer. Qu'en est-il d'applications concrètes ? Ces applications utilisent beaucoup de composants et nécessitent une infrastructure adaptée avec beaucoup plus de Deployers et de Mini Deployers. Le prototype pourra-t-il supporter une telle charge ?

Composant architecture

Lors de l'installation d'une application, une architecture de déploiement doit être définie avec la localisation de chaque composant sur l'ensemble des Deployers. La définition de cette architecture est une étape délicate car la recherche peut être compliquée. L'article [5] décrit quelques types d'algorithmes possibles pour trouver une telle architecture. Notre proposition est d'ajouter un composant directement responsable de la recherche de cette architecture au lieu de laisser l'**Instantiator** s'occuper de cette tâche.

Dans le chapitre concernant les opportunités d'améliorations 5.2, nous avions mis l'accent sur le manque de contrôle sur le déploiement. Le nouveau composant proposé ici pourrait intégrer ces notions d'exigences à atteindre et proposer des architectures orientées vers la sécurité par exemple.

Transformation du modèle

Cette idée provient du MDE [1]. On peut vouloir effectuer des transformations sur le modèle. Ces transformations permettent d'atteindre certains objectifs. Par exemple, on peut vouloir permettre à l'application d'être testée. Pour cela, des composants de logging sont ajoutés entre les composants de l'application. À chaque invocation d'une méthode d'un composant à partir d'un autre, celle-ci est sauvegardée.

[1] Model-Driven engineering

Chapitre 7

Conclusion

Dans ce travail, nous avons commencé par décrire quelques architectures de déploiement ainsi que les contextes dans lesquels elles sont utilisées.

Nous avons ensuite précisé ce que nous entendions par le déploiement et justifié pourquoi il était nécessaire d'avoir une architecture générique. C'est sur ce constat que nous avons proposé une architecture composée de deux parties, l'une effectuant le déploiement, l'autre chargée de faire tourner les composants. Nous avons alors donné quelques critiques et justifications concernant cette architecture et les points qu'il est possible d'améliorer.

Un prototype ayant été réalisé de manière parallèle avec l'architecture, nous avons tout d'abord décrit comment ce prototype avait été réalisé. Le prototype a ensuite été décrit pour réaliser le déploiement d'applications. Nous avons terminé ce chapitre par les limites du prototype.

Enfin, nous avons développé quelques pistes permettant d'améliorer l'architecture.

Le déploiement est un domaine très vaste, et nous n'avons pas pu développer certains concepts qui sont également intéressants. Nous avons proposé notre architecture car nous pensons qu'elle constitue une partie très importante du déploiement.

Nous espérons que d'autres seront intéressés par notre architecture et qu'ils y apporteront des améliorations.

Bibliographie

[1] Meriem Belguidoum and Fabian Dagnat. Toward autonomic deployment of software component.

[2] Alan Dearle, Graham Kirby, Andrew McCarthy, and Juan Carlos Diaz y Carballo. *A Flexible and Secure Deployment Framework for Distributed Applications*, pages 219–233. Lecture Notes in Computer Science. Springer Berling / Heidelberg.

[3] Areski Flissi, Jérémy Dubus, Nicolas Dolet, and Philippe Merle. Deploying on the grid with deployware. In *CCGRID '08 : Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 177–184, Washington, DC, USA, 2008. IEEE Computer Society.

[4] Fractal documentation. `http://fractal.ow2.org/specification/index.html`.

[5] Sam Malek, Marija Mikic-Rakic, and Nenad Medvidovic. An extensible framework for autonomic analysis and improvement of distributed deployment architectures. In *WOSS '04 : Proceedings of the 1st ACM SIGSOFT workshop on Self -managed systems*, pages 95–99, New York, NY, USA, 2004. ACM.

[6] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1) :70–93, 2000.

[7] Marija Mikic-Rakic and Nenad Medvidovic. Architecture-level support for software component deployment in resource constrained environments. In *CD '02 : Proceedings of the IFIP/ACM Working Conference on Component Deployment*, pages 31–50, London, UK, 2002. Springer-Verlag.

Annexe A

Description technique du prototype

Cette annexe va nous permettre de décrire les aspects techniques du prototype. Celui-ci a été développé en Java et utilise le modèle de composant Fractal.

Le prototype est composé de plusieurs applications :

- Une application **Deployer** (1200 lignes de code) : Il s'agit de l'application qui se charge du déploiement.
- Une application **Mini Deployer** (500 lignes de code) : Il s'agit de l'application qui fait tourner les composants des applications.
- Un **Naming Service** : Il s'agit d'une application qui permet d'enregistrer les Deployers qui sont connectés au système. Celui-ci permet d'associer un Deployer à un nom. Cela permet de pouvoir communiquer directement avec un Deployer en utilisant son nom sans en connaître sa localisation.
- Une **console d'accès** : Il s'agit du programme permettant de lancer l'installation d'une application à partir d'un Deployer.

Nous allons maintenant décrire la procédure d'installation d'une application. La première étape consiste en la mise en place des applications impliquées dans le déploiement :

1. Installation et lancement d'un Naming Service sur une machine.
2. Installation des deployers sur quelques machines. Pour lancer un deployer, il faut lui donner la localisation du ns (adresse ip et port) ainsi qu'un nom permettant de l'identifier. Au lancement, le deployer s'inscrit automatiquement au naming service. Par exemple, pour lancer un deployerA :

   Listing A.1 – Exemple de lancement d'un Deployer

   ```
   Deployer 192.168.1.1 12345 deployerA
   ```

3. Installation des mini deployers sur beaucoup de machines. Pour lancer un mini deployer, il faut lui donner la localisation du Naming Service (adresse ip et port), le nom du deployer auquel il est associé ainsi que son nom qui permet de l'identifier. Au lancement du mini deployer, celui-ci s'associe automatiquement à son deployer. Voici un exemple qui permet de lancer deux Mini Deployer attachés à un premier Deployer, et un autre Mini Deployer à un deuxième Deployer :

Listing A.2 – Exemple de lancement de plusieurs Deployers

```
MiniDeployer 192.168.1.1 12345 deployerA miniDeployer1A
MiniDeployer 192.168.1.1 12345 deployerA miniDeployer2A
MiniDeployer 192.168.1.1 12345 deployerB miniDeployer1B
```

Ensuite, il faut construire le package qui sera installé. Ce package doit contenir le code compilé des classes Java de l'application, les fichiers Fractal qui décrivent l'application et le fichier *package.xml* qui décrit l'application qui sera installée. Ensuite, ces fichiers doivent être placés dans une archive zip. Voici un exemple d'un fichier *package.xml* :

Listing A.3 – Exemple de fichier package.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE package SYSTEM "package.dtd">
3  <package>
4      <name>FractalTest</name>
5      <type>Fractal</type>
6      <description>adl/App.fractal</description>
7      <main>app.Main</main>
8      <constraints>
9                  <componentLocation component="client" target="
                       deployerC" />
10                 <componentLocation component="server" target="
                       deployerB" />
11                 <componentLocation component="ROOT" target="
                       deployerC" />
12     </constraints>
13 </package>
```

Enfin, pour lancer l'installation, il faut lancer la console avec comme information : la localisation du Naming Service, le Deployer sur lequel l'installation doit être lancée ainsi que le package à installer. Voici un exemple de lancement d'une installation :

Listing A.4 – Exemple de lancement d'une installation

```
Console 192.168.1.1 12345 deployerA package.zip
```

Annexe B

Architecture-Level Support for Software Component Deployment in Resource Constrained Environments

Mikic-Rakic et Medvidovic [7] détaillent plusieurs problèmes que le déploiement tente de gérer :

1. Le déploiement initial d'un système sur un nouvel hôte.
2. Le déploiement d'une nouvelle version d'un composant sur un système déjà existant.
3. L'analyse statique des effets désirés sur les modifications désirées sur le système cible.
4. L'analyse dynamique des effets sur les modifications effectuées sur le système en marche.

Les architectures logicielles fournissent une abstraction de haut-niveaux et sont décrites par des **components** qui décrivent les opérations et les états d'un système, des **connecteurs** qui décrivent les règles et mécanismes d'interaction entre composants et, enfin, des **configurations** qui définissent la topologie des **components** et des **connectors**.

B.1 Style d'architecture

Les composants échangent des messages au moyen de trois ports de communication décrits au tableau B.1 :

Il existe également des **BorderConnector** qui sont des *connectors* permettant la communication de composants entre des appareils différents :

- *Marshal* et *unmarshal* [1] les données, codes et modèle d'architecture.

[1] *Marshal* et *unmarshal* sont deux termes anglais difficilement traduisibles en français. Voila pourquoi ils apparaissent en anglais dans le texte. Les mots anglais difficilement traduisibles seront notés en italique.

Nom	**Style de message**
top	Requête
bottom	Notification
side	Peer

TABLE B.1 – Style d'architecture

Nom	Utilisé part	Description
ApplicationData	application-level	Communication lors de l'exécution
ComponentContent	meta-level	Mobilité du code et information correspondante
ArchitecturalModel	meta-level	Contient des informations pour effectuer des analyses du niveau architectural

TABLE B.2 – Différents types de messages

- Dispatchent et reçoivent les messages sur le réseau.
- Peuvent compresser les données et chiffrer les données.

B.2 Modélisation d'architecture et analyse

Il existe deux niveaux de modélisation :

1. application-level.
2. meta-level : observer et/ou faciliter différents aspects du déploiement, exécution, évolution dynamique et mobilité des composants au niveau applicatif.

Pour supporter l'utilisation de ce double niveau d'architecture, il existe une distinction entre différents types de messages (voir tableau B.2) :

Les *ComponentContent* sont utilisés dans des composants appelés *Admin Components* et qui facilitent le déploiement et la mobilité des composants applicatifs tandis que les *ArchitecturalModel* sont utilisés dans un composant *Continuous Analysis* qui permet d'analyser les modèles de l'architecture pendant l'exécution de l'application et ainsi permet de tester certains changements de l'architecture.

B.3 Système de déploiement

B.3.1 Processus de déploiement

Il exite deux types de configurations différentes : la **current configuration** qui décrit la topologie actuelle du système et la **desired configuration** qui représente la configuration qui doit être déployée. S'il existe une différence entre ces deux configurations, le processus de déploiement est initié.

Les informations concernant la configuration actuelle et la configuration désirée peuvent être stockés soit sur un seul hôte (centralisé), soit chaque sous-système possède la connaissance de sa configuration actuelle et désirée (distribué).

B.3.2 Connaissance Centralisé

Le composant *Continuous Analysis* (voir figure B.1) de l'hôte principal possède la connaissance de la configuration actuelle et désirée pour tous les sous-systèmes.

1. Le composant *Continuous Analysis* reçoit la configuration désirée, l'analyse et, si elle est valide, appelle l'*Admin Component*.
2. L'*Admin Component* envoie les composants aux hôtes cibles.
3. Réception des composants dans les différents *Admin Component/* sur chaque cible.

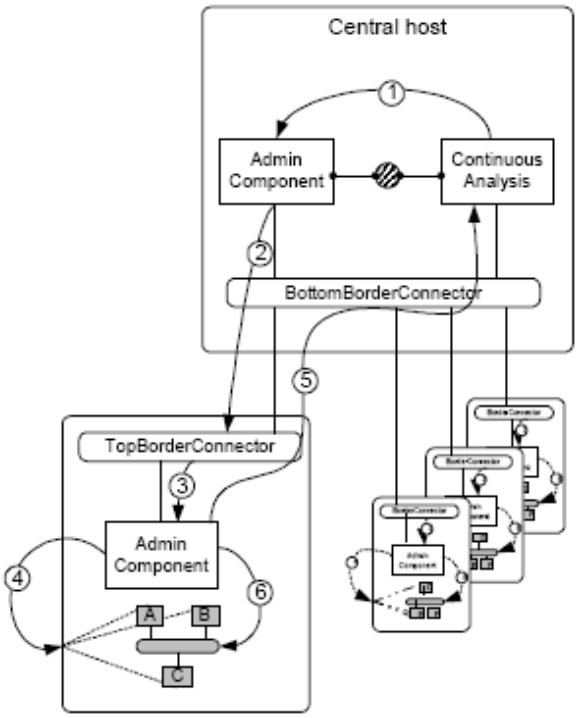


FIGURE B.1 – Connaissance centralisée (Figure provenant de [7])

4. Chaque *Admin Component* attache les différents composants reçus aux connecteurs appropriés.
5. Chaque *Admin Component* envoie un message au composant *Continuous Analysis* afin de signaler que le déploiement a été réalisé.
6. Chaque *Admin Component* lance les composants nouvellement déployés.

B.3.3 Connaissance Distribué

Chaque sous-système possède un composant *Continuous Analysis* (voir figure B.2) qui est au courant de la configuration de ces sous-systèmes. Ce composant peut effectuer diverses analyses de certaines configurations.

1. Chaque composant *Continuous Analysis* reçoit la configuration désirée pour son sous-système, l'analyse et, si elle est valide, appelle l'*Admin Component*.
2. (a) Chaque *Admin Component* demande aux autres *Admin Components* quels composants il doit installer localement.

   (b) Le *Admin Component* qui peut répondre à la requête envoie les composants désirés au périphérique demandeur.
3. Réception des composants dans les différents *Admin Components* sur chaque cible.
4. Chaque *Admin Component* attache les différents composants reçus aux connecteurs appropriés.
5. Chaque *Admin Component* envoie un message pour informer son composant *Continuous Analysis* que le déploiement a été réalisé. Le modèle de la configuration est mis à jour.
6. Chaque *Admin Component* lance les composants nouvellement déployés.

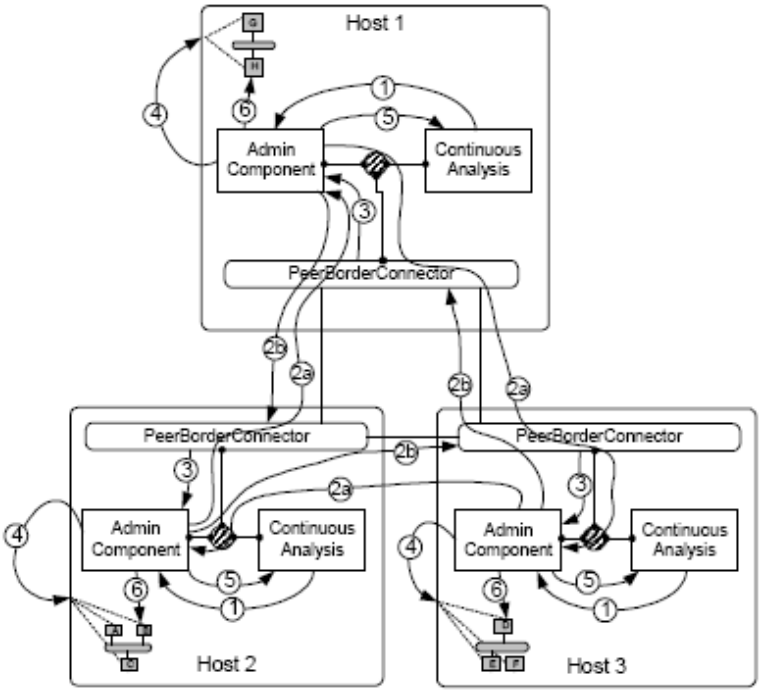


FIGURE B.2 – Connaissance distribuée (Figure provenant de [7])

B.4 Mise à jour des composants

Par l'utilisation de connecteurs multi-version (**Multi-Versionning Connectors**, voir B.3), il est possible d'avoir plusieurs versions d'un composant. Une version est marquée comme *autoritative* sur les autres et le **MVC** propage alors seulement les résultats de cette version. **MVC** enregistre les résultats de toutes les invocations aux composants multi-versionnés et les comparent aux résultats produits par la version d'autorité.

Cette autorité peut opérer soit sur l'ensemble des méthodes du composant soit sur une partie seulement. Il faut cependant que le comportement soit déterministe, c'est-à-dire qu'à chaque appel, un et un seul composant soit utilisé.

Au début, les **MVC** étaient envisagés uniquement pour la mise à jour de composants. Cependant, ils peuvent également servir lors de l'introduction d'un nouveau composant.

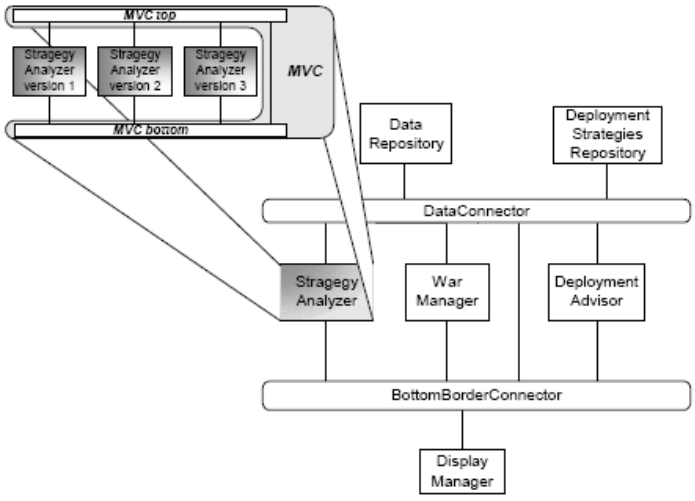


FIGURE B.3 – Exemple de MVC (Figure provenant de [7])

Annexe C

Deploying on the Grid with DeployWare

Flissi, Dubus, Dolet et Merle [3] présentent une architecture qui permet le déploiement d'applications sur un Grid. Le déploiement sur un Grid apporte certains problèmes.

Tout d'abord, la complexité dûe au grand nombre de noeuds disponibles fait que le déploiement ne peut être manuel.

Ensuite, il existe une hétérogénéité tant au niveau physique où les noeuds peuvent être différents au point de vue matériel, réseau, système d'exploitation, qu'au niveau des mécanismes de déploiement (SSH) et de transfert de fichiers (SCP, FTP). Les logiciels sont également constitués de composants qui sont hétérogènes en termes de technologies et/ou paradigmes.

Le troisième problème concerne la validation. De grand déploiements nécessitent une grande fiabilité qui est possible grâce à une validation statique avant l'exécution. En effet, si aucune validation n'est effectuée, il serait par exemple possible de déployer des applications incomplètes où certains composants seraient manquants.

Enfin, le problème de la montée en charge. Les interconnexions entre Grids créées de nouveaux problèmes, comme les limites des ressources physiques qui, par exemple, ne peuvent supporter qu'un certain nombre de sockets sur un seul noeud.

C.1 DeployWare Framework

DeployWare est un framework pour la description, le déploiement et la gestion de systèmes logiciels hétérogènes et distribués sur des Grids. A cette fin, ce framework fournit :

1. Un *DSML* (Domain-specific modeling language) pour le déploiement.
2. Une machine virtuelle qui peut interpréter des descriptions et exécuter le processus de déploiement ainsi qu'une bibliothèque de composants de bas-niveau permettant de masquer l'hétérogénéité de l'infrastructure physique.
3. Une console graphique qui permet de monitorer les systèmes déployés sur le Grid.

C.1.1 Présentation de DeployWare

Ce schéma (Fig C.1) montre les différents acteurs impliqués dans le déploiement :

1. Administrateur système qui décrit l'infrastructure cible (noeuds dans le Grid, le réseau et les protocoles).
2. Expert logiciel qui décrit les procédures de déploiement.
3. Utilisateurs finals qui fournissent le logiciel à déployer.

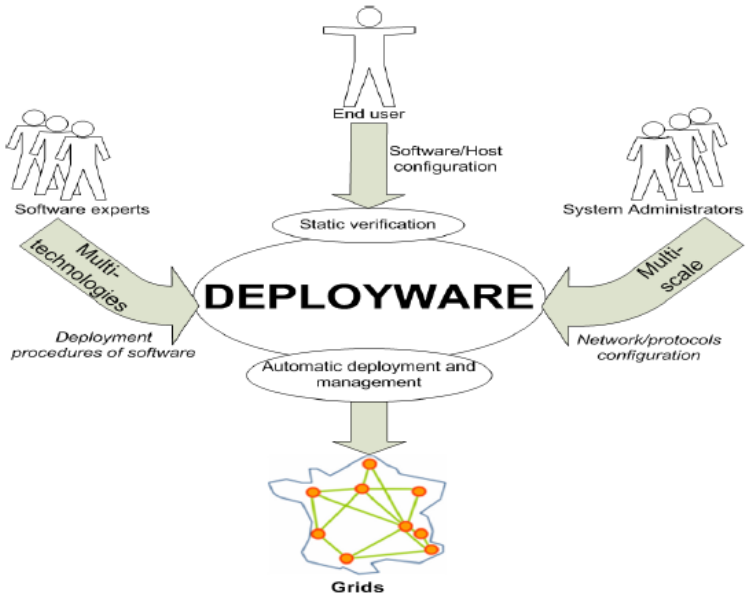


FIGURE C.1 – Présentation de DeployWare (Figure provenant de [3])

C.1.2 DeployWare Metamodel

Il y a deux parties à ce méta-modèle (Figure C.2) :

- Le package **TechnoExpert** : il définit les concepts qui sont manipulés par les experts logiciels d'une technologie donnée.
- Le package **SystemAdmin** : il est utilisé par les administrateurs/utilisateurs finals pour écrire des modèles qui représentent la configuration de leurs déploiements.

Les concepts principaux du déploiement se retrouvent également dans ce méta-modèle.

Premièrement, les concepts de **Personality** qui permet de définir un ensemble de logiciels attachés à une certaine technologie et de **SoftwareType** qui représente une abstraction du concept de logiciel.

Ensuite, il existe certaines procédures de déploiement générique (installation, configuration, ...) qui sont représentées par l'élément **Procedure**. Une **Procedure** est constituée de **Instruction**.

Il est possible de décrire des propriétés génériques qu'il est possible de configurer et qui sont représentés par le concept de **Property**.

Enfin, une **SoftwareInstance** représente un logiciel qui est déclaré à l'exécution par les utilisateurs.

Les modèles sont validés avant l'exécution du déploiement. Pour cela, plusieurs conditions doivent être satisfaites :

- Toutes les dépendances entre logiciels doivent être vérifiées.
- Pour chaque procédure, la procédure opposée doit exister (installation et désinstallation).
- Vérification à l'intérieur des procédures, elles doivent être symétriques. Par exemple, lors du lancement de l'application, un serveur peut être lancé et doit être coupé à la fin de l'application.

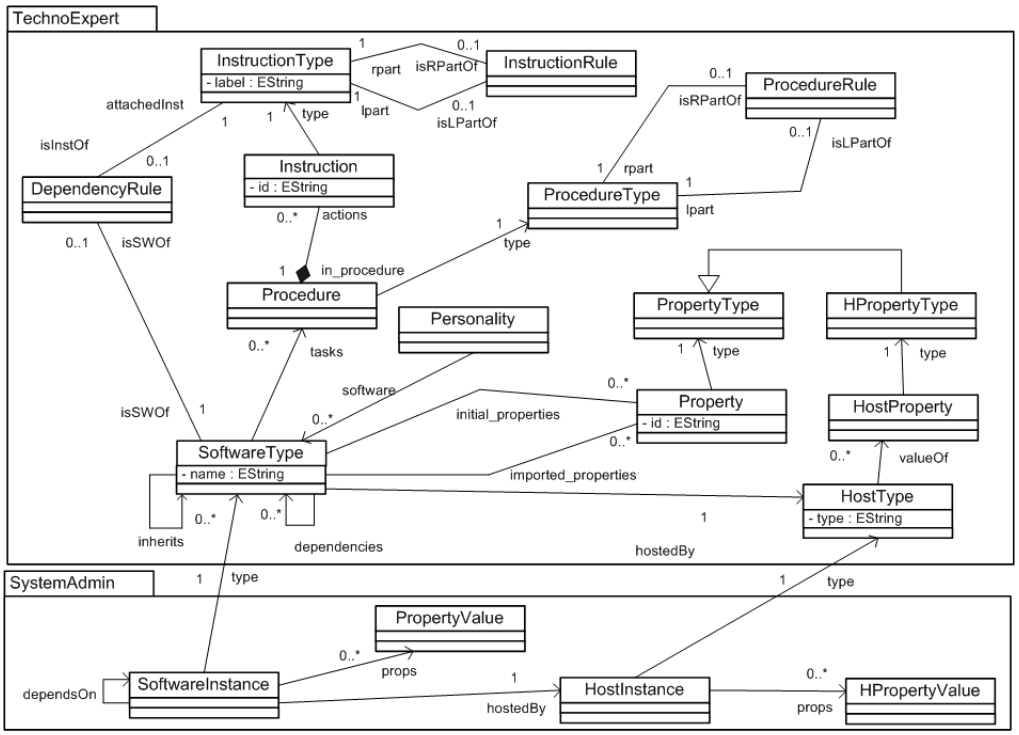


FIGURE C.2 – DeployWare Metamodel (Figure provenant de [3])

- Vérification sur les ports et systèmes de fichiers pour éviter les conflits de ressources.

Les modèles dans DeployWare proviennent de fichiers de descriptions DeployWare qui utilisent une syntaxe rigoureuse. Cette syntaxe permet de décrire les systèmes à installer et de cacher la complexité aux utilisateurs finals et aux administrateurs. Ils n'ont en effet juste qu'à décrire l'infrastructure du logiciel au lieu de programmer le processus de déploiement.

Ce fichier de description DeployWare est constitué d'une partie qui décrit l'infrastructure (*hosts*) et d'une partie qui spécifie le logiciel à déployer sur quels hôtes (*software*).

C.2 DeployWare Runtime

Une machine virtuelle (Figure C.3) exécute les descriptions et organise le processus de déploiement. Le système est décrit au moyen de **FDF** (Fractal Deployment Framework).

Il est possible de distinguer deux niveaux. Premièrement, le niveau des composants logiciels qui constituent un semble de composants qui réifient le logiciel de haut-niveau à déployer. Le second niveau est une bibliothèque de composants qui abstraient les mécanismes de déploiement de bas-niveaux ce qui permet de masquer l'hétérogénéité de l'infrastructure physique.

Un logiciel est représenté comme une composition de composants (figure C.4) :

Le composant composite **properties** contient les propriétés configurables du logiciel, le composant **dependencies** contient les références aux autres logiciels. Les différentes procédures (**install**, **start**, ...) sont également des composants composites et où les instructions sont également des composants qui utilise la couche *Deployment components* pour réaliser les tâches de déploiement élémentaires.

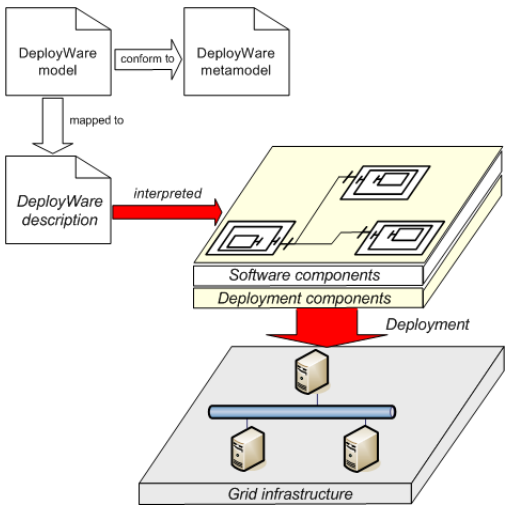


FIGURE C.3 – Machine virtuelle (Figure provenant de [3])

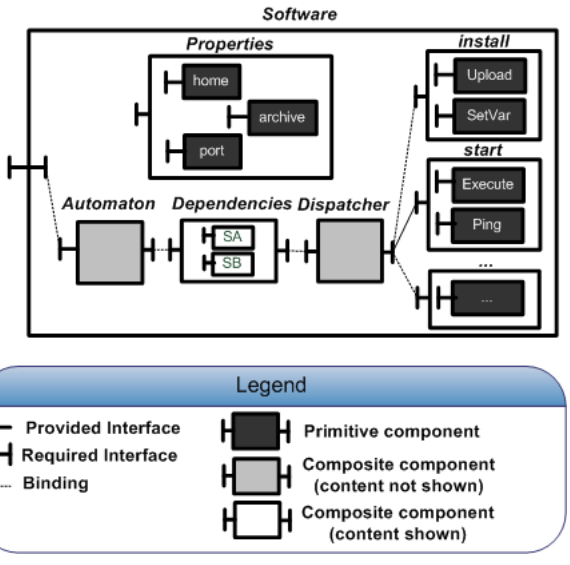


FIGURE C.4 – Logiciel est représenté comme une composition de composants (Figure provenant de [3])

Dans le cas où le nombre de noeuds est très important, il est possible de démarrer plusieurs **FDF** sur différents noeuds (figure C.5). Chaque serveur **FDF** est en charge du déploiement d'une partie du système. L'installation de ces serveurs ce fait en suivant le même mécanisme.

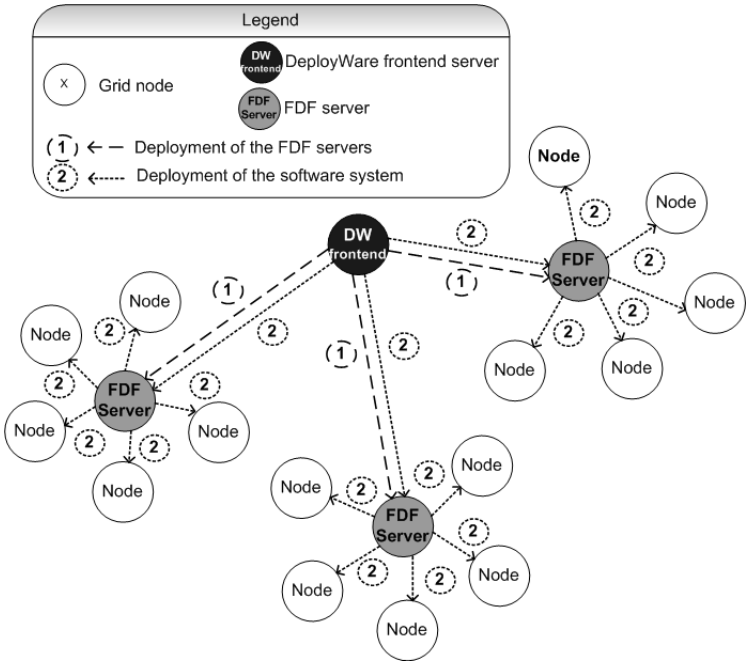


FIGURE C.5 – Distribution de la charge de déploiement (Figure provenant de [3])

Annexe D

An Extensible Framework for Autonomic Analysis and Improvement of Distributed Deployment Architectures

Malek, Mikic-Rakic et Medvidovic [5] ont décrit plusieurs problèmes que les systèmes de déploiement doivent résoudre. Le première problème concerne le grand nombre de paramètres qui influencent la sélection de l'architecture de déploiement. De plus, ces paramètres ne sont pas tous connus à la définition du système et peuvent varier lors de l'exécution. Enfin, il est difficile de trouver une architecture optimale de part le grand nombre de possibilités.

Ils ont pour cela développé une méthodologie qui leur permet d'améliorer la disponibilité du système par un monitoring actif du système, une estimation de l'amélioration de l'architecture de déploiement et d'un redéploiement d'une partie ou de la totalité de l'architecture de déploiement.

Ils ont également développé un framework D.1 afin d'analyser et d'améliorer le déploiement distribué d'architectures.

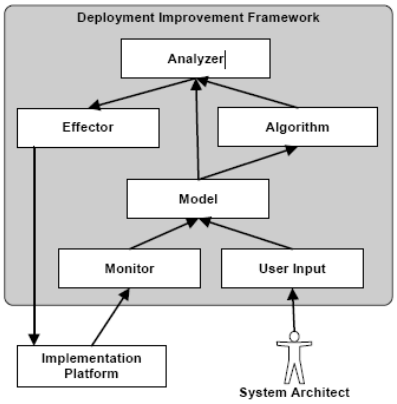


FIGURE D.1 – Design du framework (Figure provenant de [5])

Le **Model** maintient une représentation de l'architecture de déploiement du système. Cet élément est constitué des hôtes qui sont les différents noeuds sur lesquels il est possible d'installer des composants, les composants eux-mêmes, les liens physiques entre les hôtes ainsi que les liens logiques entre les composants. Ces différents éléments peuvent avoir un

certain nombre de paramètres (comme par exemple la mémoire vive disponible).

L'élément **Algorithm** reçoit comme entrée un objectif et un sous-ensemble du modèle du système. Sur cette base, un algorithme recherche une architecture de déploiement qui satisfait l'objectif. Cet algorithme peut également rechercher une architecture de déploiement qui satisfait plusieurs objectifs.

L'**Analyzer** est un algorithme qui utilise les résultats obtenus par les algorithmes et le modèle pour déterminer une séquence d'actions pour satisfaire les objectifs globaux du système. Lorsqu'il y a de multiples objectifs, il n'est cependant pas possible de toujours tous les satisfaire. L'**Analyzer** est également capable de modifier le comportement du framework en fonction des paramètres du système et peut être amené à enregistrer des fluctuations des objectifs lors de l'exécution du système.

Le **Monitor** est un élément qui est associé à chaque paramètre du système et permet de déterminer leurs valeurs à l'exécution. Chaque **Monitor** possède une implémentation dépendante de la plate-forme qui est chargée de récolter les données et d'une une implémentation indépendante de la plate-forme qui vérifie que les données sont suffisamment stables pour être envoyées au modèle.

L'**Effector** est également composé de deux parties : une implémentation dépendante de la plate-forme qui permet de s'attacher dans la plate-forme pour effectuer le redéploiement de composants logiciels et une implémentation indépendante de la plate-forme qui reçoit les instructions de redéploiement de l'**Analyzer** et qui coordonne le processus de redéploiement. Ce composant peut parfois être amené à effectuer par exemple des tâches de buffering.

Le **User Input** permet à l'architecte du système de fournir des paramètres du système qui ne peuvent pas être facilement mesurables. Il peut également s'agir de contraintes sur les architectures possibles (comme la localisation de composants par exemple).

Ce framework peut être instancié de plusieurs manières différentes dont voici les principales :

D.1 Instanciation centralisée

Dans cette instanciation (Figure D.2), il existe un *Master Host* qui gère toutes les étapes et un *Slave Host* qui suit les instructions du *Master*. Cette type d'architecture est assez simple à réaliser, mais nécessite que le *Master* ait une connaissance globale du système.

Il existe trois types d'algorithmes centralisés :

Exact : essaie tous les déploiements possibles et sélectionne celui ayant une disponibilité maximale et qui satisfait les contraintes ($O\left(k^n\right)$).

Stochastique : ordonne tous les hôtes et tous les composants de manière aléatoire. Ensuite, en suivant cet ordre, il assigne le plus de composants possibles à chaque hôte en respectant sa capacité et en vérifiant que toutes les contraintes sont satisfaites ($O\left(n^2\right)$).

Avala : assigne les composants logiciels de manière incrémentale aux hôtes physiques. A chaque étape de l'algorithme, le but est de choisir l'assignation qui contribue de

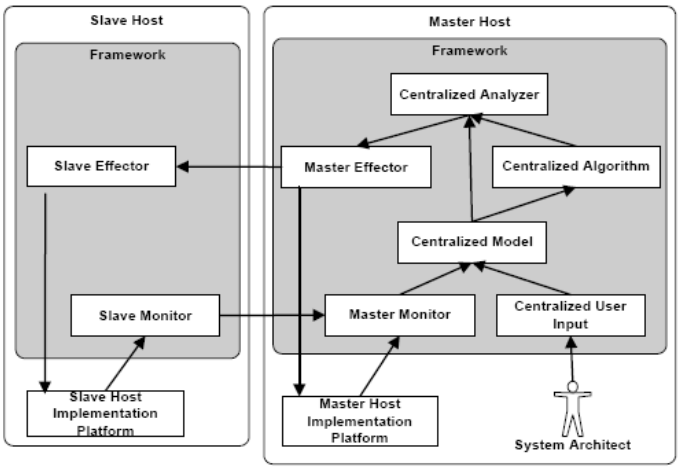


FIGURE D.2 – Instanciation centralisée (Figure provenant de [5])

manière maximale à la fonction de disponibilité, en sélectionnant le meilleur hôte et le meilleur composant logiciel ($O\left(n^3\right)$).

D.2 Instanciation décentralisée

Dans cette instanciation (Figure D.3), il n'y a plus de maître ni d'esclave, juste des hôtes qui communiquent entre eux pour réaliser les objectifs du déploiement. Chaque hôte possède des composants locaux (*Effector* et *Monitor*) qui se chargent uniquement du monitoring et du redéploiement sur l'hôte où ils se trouvent. Chaque hôte possède un *Decentralized Algorithm* qui se synchronise avec les autres hôtes afin de trouver une solution commune, de même que les *Decentralized Analyzer*.

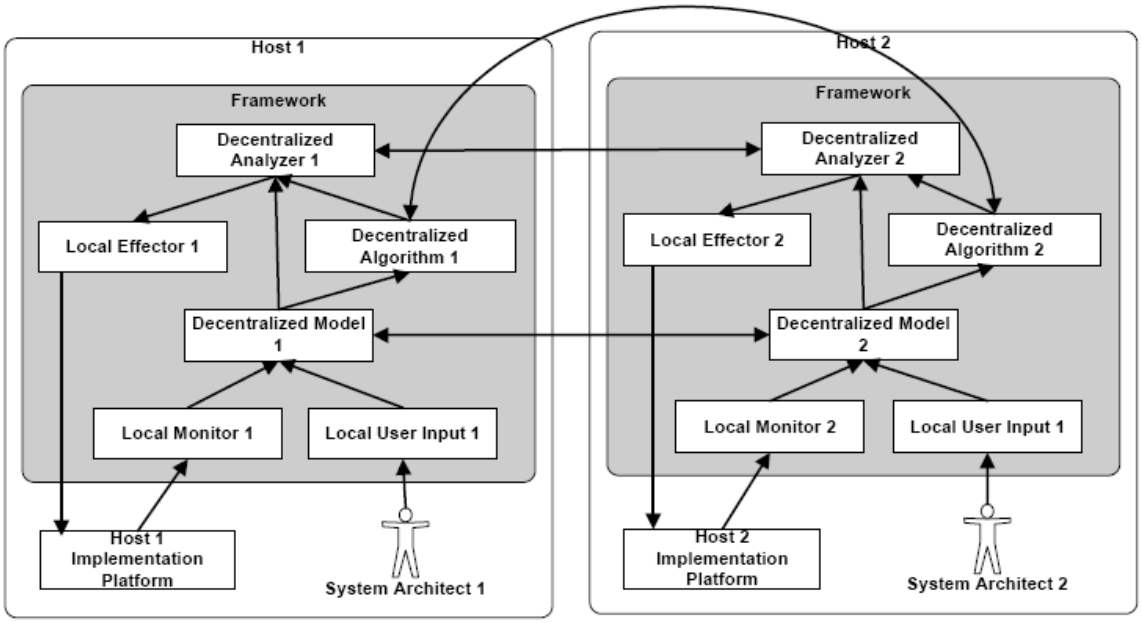


FIGURE D.3 – Instanciation décentralisée (Figure provenant de [5])

Annexe E

Toward autonomic deployment of software component

Belguidoum et Dagnat [1] discutent de l'automatisation du déploiement de composants logiciels et proposent un modèle générique E.1 constituées de plusieurs éléments.

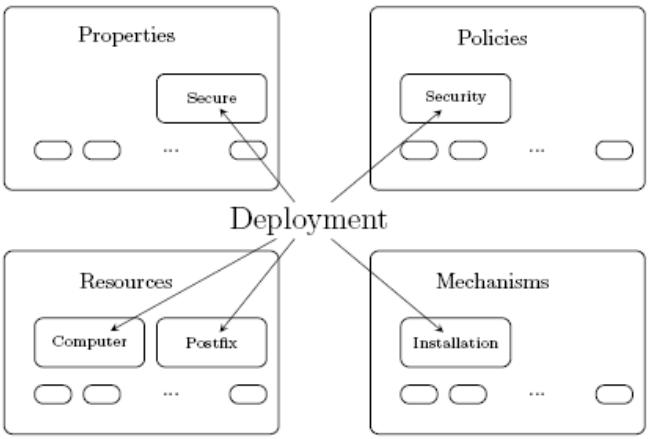


FIGURE E.1 – Modèles de déploiement (Figure provenant de [1])

Tout d'abord, les **resources** qui représentent les entités qui sont gérées lors du déploiement (le composant déployé et le système cible). Les différentions actions qui ont un effet sur ces ressources lors du déploiement sont appelés **mechanisms**. Les **policies** sont des descriptions de la manière dont les choix des **mechanisms** sont gérés. Ces trois modèles peuvent se voir affecter des **properties**.

Les modèles de déploiement sont récursifs, ce qui signifie que des entités sont composées d'autres entités du même type.

E.1 Le méta-modèle

Ce méta-modèle E.2 est constitué de deux parties. La première, représentée par des cadres blancs, représente les quatre modèles de déploiement, à savoir, *Resource*, *Mechanism*, *Property* et *Policy*. La seconde partie, représentée quant à elle par des cadres gris, représente le raisonnement de l'application.

Une **ressource** peut représenter le système cible ou une entité déployée. Le contexte **description** représente toutes les entités qui constituent la **ressource**.

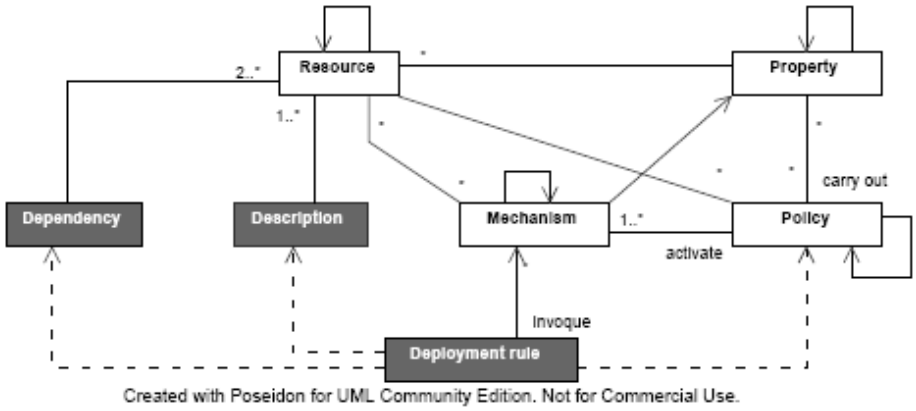


FIGURE E.2 – Méta-modèle de déploiement (Figure provenant de [1])

Dependencies représente la relation entre les ressources nécessaires et les ressources fournies. Les dépendances ne sont pas toute de même nature et il est possible de les diviser en trois groupes :

1. dépendance obligatoire : pour qu'une ressource soit capable d'effectuer son travail, elle a absolument besoin d'une autre ressource.

2. dépendance optionnel : une ressource peut fonctionner avec ou sans une autre ressource. C'est le cas par exemple d'une ressource responsable du logging d'informations.

3. dépendance négative : une ressource ne peut fonctionner si une autre ressource est présente.

Un **mechanism** est une action qui peut être exécutée pendant le déploiement et qui a un effet sur le système cible. Elle peut représenter des actions simples ou des actions plus complexes. Les **mechanisms** sont choisis par les **policies**.

Policy guide le choix des **mechanisms**. Ce choix est basé sur les buts de déploiement (**properties**) et les conditions de déploiement. Une **Policy** peut être basique ou composée et est souvent décrite en utilisant le modèle **ECA** (*Event-Condition-Action*).

Une **property** décrit les caractéristiques d'une **resources**, **mechanisms** ou **policies**. Le modèle des propriétés permet au système de déploiement d'utiliser des propriétés nouvellement définies et de renforcer le partage de la même sémantique de propriétés aux travers des entités participant au déploiement.

Les **deployment rules** vérifient la faisabilité du déploiement en accord avec les exigences de l'environnement et des contraintes des entités.

E.2 Architecture de déploiement

Avant tout déploiement, chaque entité (ressources, mécanismes, politiques et propriétés) doit être abstraite en une description réelle.

Le contexte physique ainsi que les composants physiques sont interprétés dans le monde formel en tant que respectivement *description du contexte* et *dépendances* et sont exprimés dans un langage logique.

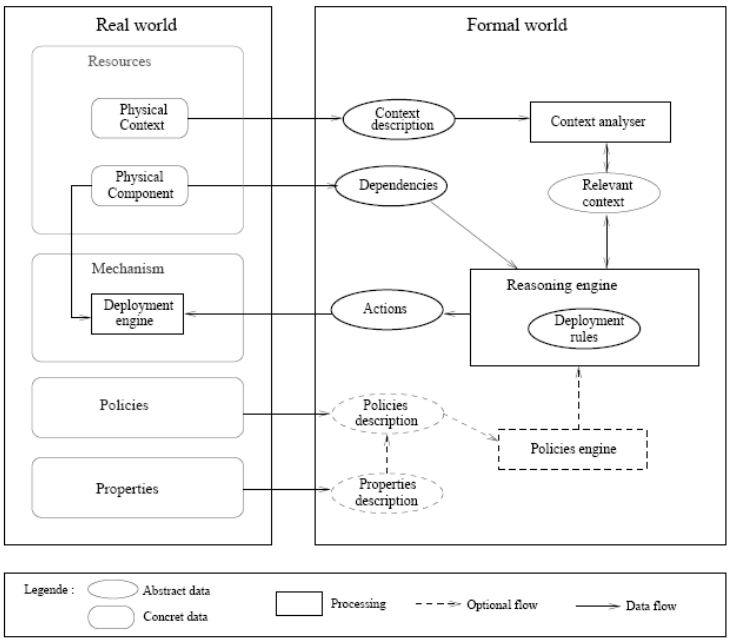


FIGURE E.3 – Architecture de déploiement (Figure provenant de [1])

Ensuite, l'analyser calcule le contexte et les exigences les plus pertinentes pour le déploiement.

Sur base de la description du contexte et des dépendances ainsi que de règles de déploiement, le moteur de raisonnement peut vérifier si le déploiement est possible. Le déploiement est possible si toutes les dépendances sont vérifiées.

Le moteur de raisonnement peut alors calculer l'effet de déploiement sur le contexte et générer des actions abstraites qui, lorsqu'elles seront exécutées dans le monde réel, permettront au déploiement de s'effectuer. Les actions abstraites sont transformées en mécanismes par le moteur de déploiement, mécanismes qui représentent les différentes étapes du déploiement concret.

Les politiques et les propriétés sont interprétées afin de pouvoir être utilisées par le moteur de politiques qui pourra influencer le moteur de raisonnement, en choisissant par exemple quels actions seront effectuées pour réaliser le déploiement.

Lors du déploiement, des problèmes peuvent apparaître comme par exemple une défaillance matériel. Le déploiement est alors stoppé et le contexte n'est pas mis à jour. À l'inverse, lorsqu'un déploiement se termine sans problème, le contexte physique est mis à jour, ainsi que la description du contexte.

E.3 Comment décrire le déploiement

E.3.1 Les dépendances

Les logiciels qui sont déployés dans une telle architecture sont formés de composants interconnectés et c'est cette inter-connexion qui représente l'architecture du logiciel. Dans cette architecture, il existe plusieurs types de dépendances :

- dépendance **obligatoire** : un composant nécessite un autre composant pour fonctionner.

- dépendance **optionnel** : un composant peut utiliser un autre composant afin de fournir des services optionnels.
- dépendance **négative** : un composant ne peut pas fonctionner si un autre composant est présent.

E.3.2 Description du contexte

Le contexte représente les composants déjà installés. Pour représenter ce contexte, il est nécessaire de l'approximer. Cet article propose de le représenter comme ceci :

1. un environnement ε stockant les valeurs de certaines variables.
2. un ensemble $\mathcal{C}$ contenant les composants installés et les services qu'ils fournissent $\mathcal{P}_s$, les services interdits $\mathcal{F}_s$ et les composants interdits $\mathcal{F}_c$.
3. un graphe de dépendance $\mathcal{G}$ contenant les dépendances. Dans ce graphe, une arête entre n_1 et n_2 signifie que n_2 exige n_1. De plus, chaque arête possède un label pour savoir si la dépendance est obligatoire ou optionnelle.

E.3.3 Moteur de raisonnement

Le moteur de raisonnement représente l'ensemble des règles de déploiement qui peuvent être utilisées par le système. Les phases d'installation et de désinstallation ont été abstraites afin de pouvoir donner des règles permettant de s'assurer de la bonne exécution de ces étapes.

La phase d'installation peut être divisée en deux phases dont le schéma est représenté à la figure E.4.

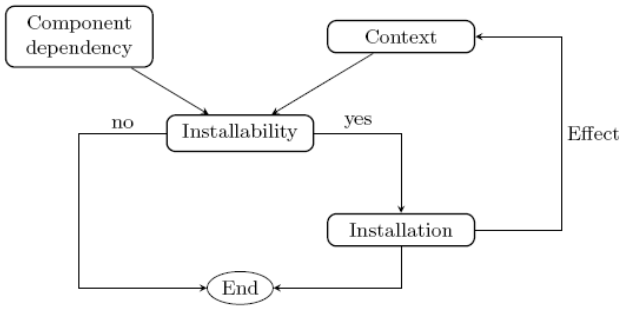


FIGURE E.4 – Phase d'installation (Figure provenant de [1])

La première phase consiste à vérifier si l'installation est possible (*installability*). Cette vérification utilise le graphe de dépendances ainsi que le contexte et utilise des règles de vérification pour s'assurer de la validité des dépendances quant elles seront ajoutées dans le contexte.

Si l'installation est possible, cette seconde étape est effectuée en utilisant les règles d'installation. Elle calcule les effets de l'installation sur le contexte et l'utilise pour mettre à jour le contexte une fois que l'installation concrète a été effectuée. Cette modification du contexte est constituée des nouveaux services disponibles, des nouveaux services interdits, des nouveaux composants interdits ainsi que des nouvelles dépendances entre composants (avec modification du graphe de dépendances).

La phase de désinstallation est également exécutée en deux phases.

La première phase consiste à vérifier si la désinstallation est possible en s'assurant qu'aucun service fourni par les composants ne soit requis par d'autres composants.

Ensuite, si la désinstallation est possible, la seconde étape se charge de calculer l'effet de la désinstallation qui consiste en la suppression du composant du contexte et des relations aux services que ce composant fournit dans le graphe de dépendance.

Annexe F

A Flexible and Secure Deployment Framework for Distributed Applications

Il existe plusieurs exigences pour un service de déploiement flexible d'après Dearle [2] :

1. Une description architecturale des composants logiciels, des hôtes sur lesquels ils doivent être exécutés et les interconnexions entre eux.
2. la possibilité de transformer la description de l'architecture pour obtenir un déploiement possible ayant un ensemble précis de composants nécessite :
    - la possibilité d'installer et d'exécuter du code sur les hôtes distants.
    - un mécanisme de sécurité pour empêcher des personnes non autorisées de déployer et exécuter des agents nocifs et d'empêcher les composants déployés d'interférer entre eux, accidentellement ou par malveillance.
3. support de l'implémentation de composants utilisant des standards de langage de programmation et de modélisations de programmes appropriés.
4. la possibilité pour les composants de s'interfacer avec des composants *off-the-shelf* déjà déployés.

L'architecture présentée [2] introduit de nouveaux domaines de sécurités, appelés *thin servers*, qui peuvent être placés à l'intérieur d'un réseau existant. Cette architecture tente de fournir les différents points suivants :

1. *mécanismes pour déployer du code et des données dans un environnement distribué.*
2. *abstractions des attributs spécifiques des noeuds fournissant un environnement homogène pour les composants déployés.*
3. *mécanismes de liaisons sûrs afin que les composants déployés soient capables d'accéder aux données ainsi qu'aux services sur les noeuds où ils sont présents.*
4. *mécanismes pour décrire et déployer des applications distribuées constituées par des composants tournant sur de multiples noeuds.*
5. *la possibilité de faire évoluer la topologie des applications et des composants déployés.*
6. *mécanismes de sécurité qui permettent à un large éventail de politiques d'être implémentées.*

F.1 Cingal Computational Model

Cingal [1] est le nom du framework de déploiement sur lequel l'architecture se base. Dans ce framework, chaque *thin server* (Fig F.1) peut fournir les éléments suivants :

- un port sur lequel un *bundle* de code et de données peut être envoyé pour être exécuté.
- mécanisme d'authentification, empêchant du code non autorisé d'être exécuté.
- un stockage de contenu adressable.
- des *binders* de noms symboliques pour les données et processus.
- un ensemble extensible d'environnements d'exécution appelé *machines*.
- une communication inter-machine basée sur des canaux de communication asynchrones.
- un système permettant de contrôler les accès aux données, machines et liaisons.

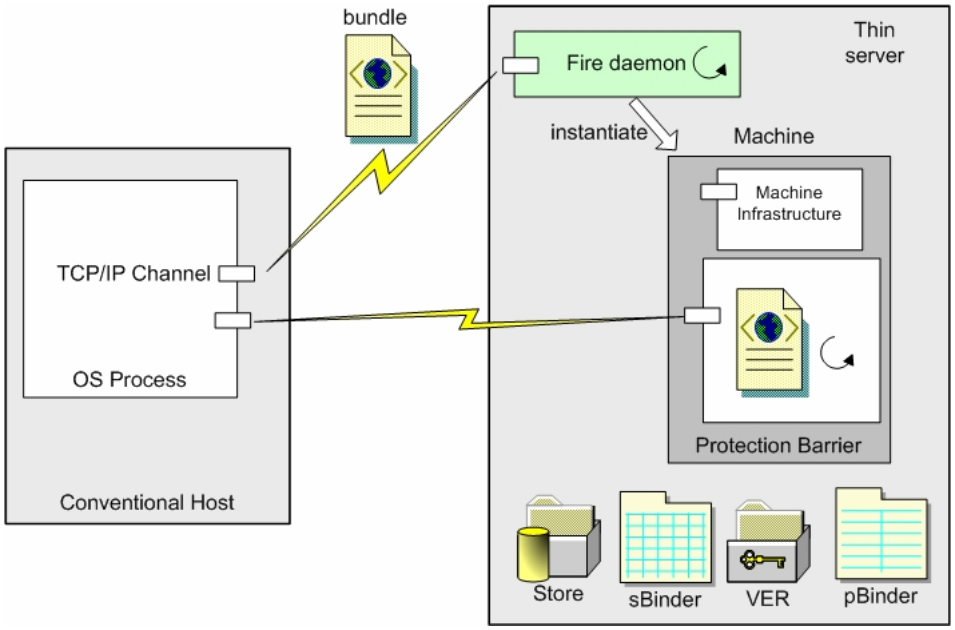


FIGURE F.1 – Modèle de Cingal (Figure provenant de [2])

L'infrastructure d'un *thin server* inclut un certain nombre de services qui peuvent être utilisés par les *bundles* exécutés à l'intérieur des machines :

- Le **store** : il fournit un moyen de stockage adressable.
- Le **store binder** et le **process binder** : ils fournissent le stockage des liaisons entre machines.
- Le **valid entity repository (VER)** : il répertorie les entités valides avec les certificats de sécurité associés.

[1] *Computation IN Geographically Appropriate Locations*

Chaque machine possède au moins deux canaux de communication. Le premier canal, appelé le **machine channel** qui est utilisé pour communiquer avec l'infrastructure de la machine. Le deuxième canal, appelé le **default channel** qui est utilisé pour communiquer avec le *bundle* tournant à l'intérieur de la machine. Ce canal est normalement utilisé pour des diagnostics et le passage de paramètres.

Ces canaux sont gérés par un composant appelé *connection manager*, situé à l'intérieur de la machine. Il maintient une liste des noms des canaux et le mapping résultant. Ce mapping peut être manipulé par d'autres machines au travers du *machine channel*.

F.2 Déploiement d'applications

Cingal fournit une infrastructure permettant de déployer des composants. Il est cependant nécessaire d'ajouter certains éléments afin de permettre de décrire une architecture distribuée ainsi que de pouvoir déployer des composants à partir de leur description. Il est donc nécessaire d'avoir un langage de description pour ces composants, un moteur de déploiement qui va lire ces descriptions et les transformer en instructions de déploiements.

Cette architecture propose différents outils pour réaliser un déploiement :

- Les **installers** : ils installent un nombre arbitraire de *bundle* dans le *store* du *thin server* de destination.
- Les **runners** : ils lancent l'exécution d'un certain nombre de *bundles* précédemment installés.
- Les **wirers** : ils créent les connexions entre les paires de composants en utilisant le mécanisme des canaux.

Chaque composant dans un *thin server* est toujours dans un certain états suivant l'outil de déploiement précédemment utilisé. Tout d'abord, l'état **installed** qui est l'état après qu'un composant soit installé dans le *store* du *thin server*, l'état **running** quand le composant a été lancé et a commencé son travail et l'état **wired** quand le composant a été lancé et que les liaisons ont été effectuées avec les autres composants (état *connecté*).

La figure F.2 montre les interactions possible de ces états.

FIGURE F.2 – Les interactions entre les outils

A l'état initial, l'état d'un composant passe de *installé* → *lancé* → *connecté*. Il existe plusieurs modifications sur le système qui peuvent modifier ces états :

- Lorsque des connexions entre composants sont modifiés. Il s'agit donc juste de la topologie des composants qui change. Les composants concernés passent alors de l'état *connecté* → *lancé* → *connecté*.
- Lorsque des composants sont déplacés sur d'autres *thin servers*. Il s'agit d'une modification plus importante que la précédente et les composants doivent être arrêtés avant d'être déplacés. Ils passent alors de l'état *connecté* → *lancé* → *installé* et sont ensuite déplacés sur le nouveau *thin server*. Sur le nouveau *thin server*, ils repassent de l'état *installé* → *lancé* → *connecté*.

Les trois outils de base (*installers*, *runners* et *wirers*) sont en fait des *bundles* configurés avec le *bundle* à installer. Ils peuvent donc être installés comme n'importe quel *bundle*.

Maintenant que les outils nécessaires au déploiement ont été présentés, il convient de décrire le processus d'installation d'une application.

1. Lecture du DDD par le moteur de déploiement.
2. Récupération des différents *bundles* à installer.
3. Configuration des installeurs pour chaque *bundle*.
4. Lancement des installeurs. Un installeur extrait le *bundle* à installer et l'ajoute au *Store*. Il retourne alors un rapport au moteur de déploiement avec les clés dans le *Store* du bundle installé.
5. Configuration des *runners* pour chaque *bundle* à lancer.
6. Lancement des *runners* (Figure F.3). Chaque *runner* est envoyé sur le *thin server* sur lequel il doit lancer un *bundle* (1). Le *thin server* instancie alors le *runner* (2) qui peut alors extraire le *bundle* à exécuter du *Store* (3) et le lance (4). Le *runner* retourne alors un rapport au moteur de déploiement (6) avec la liste des connecteurs du *bundle*. Cela permet au moteur de déploiement de communiquer directement avec le *bundle*.
7. Configuration des connecteurs pour chaque connexion à réaliser.
8. Lancement des connecteurs (Figure F.4). Chaque connecteur est envoyé (de manière arbitraire) sur le *thin server* sur lequel il doit effectuer une connexion (1). Le *thin server* instancie alors le connecteur (2). Celui-ci communique avec le *bundle* local à connecter au travers du canal machine (3). Le connecteur envoie ensuite un *bundle* contenant les informations de connexion au deuxième *thin server* (4). Celui-ci instancie le *bundle* (5) et communique au travers du canal machine pour effectuer la connexion (6).
9. Quand tous les connecteurs ont accompli leur tâche, le processus de connexion est terminé ainsi que le déploiement. Le résultat est visible à la Figure F.5.

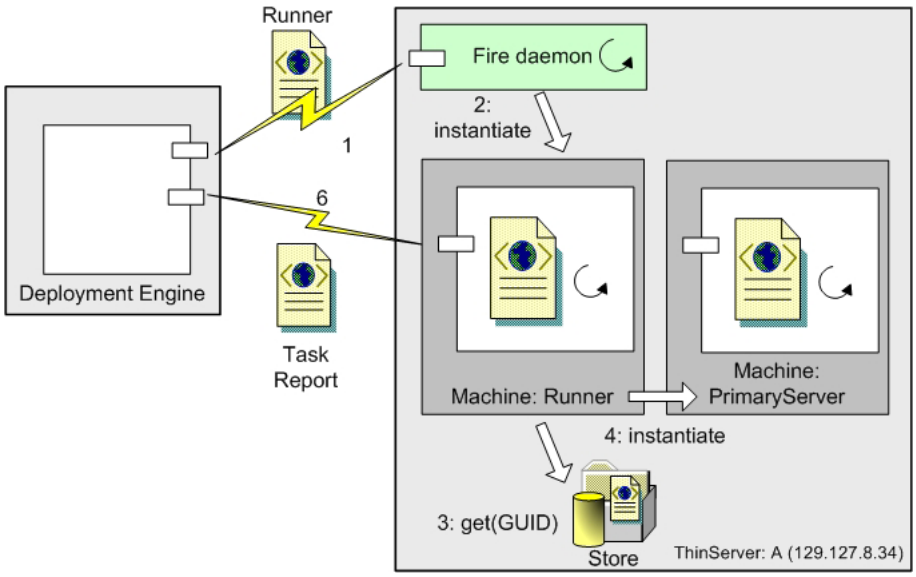


FIGURE F.3 – Lancement d'un *bundle* (Figure provenant de [2])

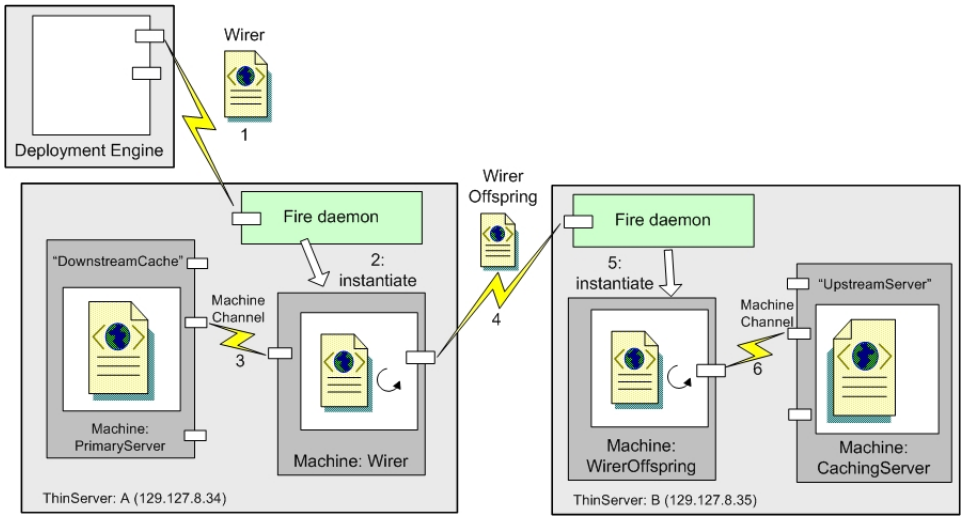


FIGURE F.4 – Processus de connexion (Figure provenant de [2])

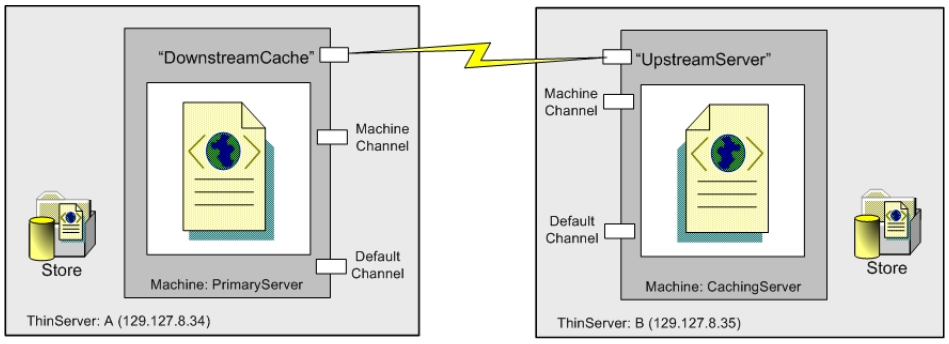


FIGURE F.5 – Application lancée et connectée (Figure provenant de [2])

Annexe G

Cingal - Exemple de bundle d'installation

Listing G.1 – Exemple bundle installation

```
<BUNDLE>
    <AUTHENTICATION entity='197301m7wWwPxX9..EySLGU'
                    signature='kUdzrv..fjq3fqsd' />
    <CODE entry='uk.ac.stand.cingal.Installer' type='java'>
        <CLASS name='uk.ac.stand.cingal.Installer'>
            5ledfc34Usj390CKLcsdfJKL...
        </CLASS>
    </CODE>
    <DATA>
        <DATUM id='urn:cingal:a222jdjd2s'>
            <BUNDLE>
                <AUTHENTICATION entity='1973012..91509'
                                signature='DQowffqsd3..
                                   fdisqDfjds48' />
                <CODE entry='Server' type='ajva'>
                    <CLASS name='Server'>
                        5fdckJFKLDcdfskdjUFIOU9587dfscjU7...
                    </CLASS>
                </CODE>
                <DATA />
            </BUNDLE>
        </DATUM>
        <DATUM id='ToDoList'>
            <TODOLIST>
                <TASK guid='urn:cingal:aEcncdeEe' type='INSTALL'
                    >
                    <DATUM id='PayloadRef'>
                        urn:cingal:a222jdjd2s
                    </DATUM>
                </TASK>
            </TODOLIST>
        </DATUM>
    </DATA>
</BUNDLE>
```

Annexe H

Description de Fractal

Fractal est la bibliothèque utilisée dans le prototype et comme modèle de composants. Il convient dès lors d'en faire une rapide présentation.

Une application Fractal est constituée de composants Fractal et d'une description des relations entre ces composants. Un composant Fractal est un objet (Java par exemple) fournissant un certain degré de contrôle.

Chaque composant peut définir des interfaces qui seront des points d'accès avec d'autres composants. La communication entre composants ce fait au moyen de ces interfaces et d'une opération de liaison appelé *bindings*.

Un composant peut soit être de type **composite**, ce qui signifie qu'il est composé de différents composants. Un composant **composite** peut donc contenir d'autres composants **composites**. On peut voir ce composant comme une boite blanche. Un composant peut également être un composant **primitif**, il s'agit alors d'un composant de base. On peut voir ce composant comme une boite noire.

Les interfaces possèdent un nom afin de les distinguer des autres interfaces ainsi qu'une signature. Il existe des interfaces de deux types : une interface client qui envoie des invocations d'opérations et une interface serveur qui reçoit ces opérations. Les interfaces décrivent les services offerts ou requis par un composant.

L'architecture d'une application peut être décrite par plusieurs moyens. L'architecture du prototype est décrite au moyen de fichiers ADL qui décrivent les composants présents ainsi que la manière dont ils sont connectés ensemble. Fractal fournit une bibliothèque qui permet de créer l'application sur base de sa description et permet alors de la lancer. L'architecture explicite ainsi que la séparation stricte entre interface et implémentation permet de faciliter l'implémentation de systèmes complexes.

Il existe une convention graphique permettant de représenter des architectures en Fractal. Les composants sont représentés par des boîtes avec leur nom à l'intérieur. Les interfaces qui sont sur le côté gauche des composants représentent les interfaces clientes, tandis que celles sur le côté droit représentent les interfaces serveurs. Il existe également des interfaces de contrôle placées sur le haut des composants. Chaque interface possède un nom afin de pouvoir l'identifier.